Unidad 4

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 8

Colecciones

En este capítulo

- Interfaces para colecciones
- Las interfaces IEnumerable e IEnumerator con el ciclo foreach
- Clases de colecciones
- Diccionarios
- Colecciones genéricas
- Interfaces para colecciones genéricas
- Clases genéricas para colecciones
- Covarianza y contravarianza

Interfaces para colecciones

Antecedentes

Las colecciones son un concepto intuitivo implementado desde los comienzos de los lenguajes orientados a objetos en diversas clases, algunas de uso interno y otras provistas para ser utilizadas por los programadores. Este concepto en realidad es una mejora respecto de los vectores de objetos para generar diversos tipos de agrupamientos de los mismos.

Manejar objetos agrupados implica que debe haber al menos una clase que los contenga y que brinde algún algoritmo de gestión para administrarlos. Dichos algoritmos existen en diversas formas y muchas de ellas fueron implementadas en distintas clases. Siguiendo el espíritu original que llevo al manejo de estructuras de datos, se diseñaron clases que manejan los mismos tipos de algoritmos (lamados en ciencias de la computación “algoritmos fundamentales”), nada más que en lugar de utilizar datos, se usan objetos.

En la primeras implementaciones, como la versión 1.0 del Framework. NET se definió un amplio conjunto de clases de colección que puede contener cualquier tipo de objeto.

Las clases de colección basadas en objetos se definen en el espacio de nombres System.Collections. Este espacio de nombres también define un conjunto de interfaces que especifican las operaciones comunes que estas clases de colección implementan. Esta técnica se basa en separar la funcionalidad básica de los algoritmos en las interfaces para que las clases que los implementen compartan dicha funcionalidad en común.

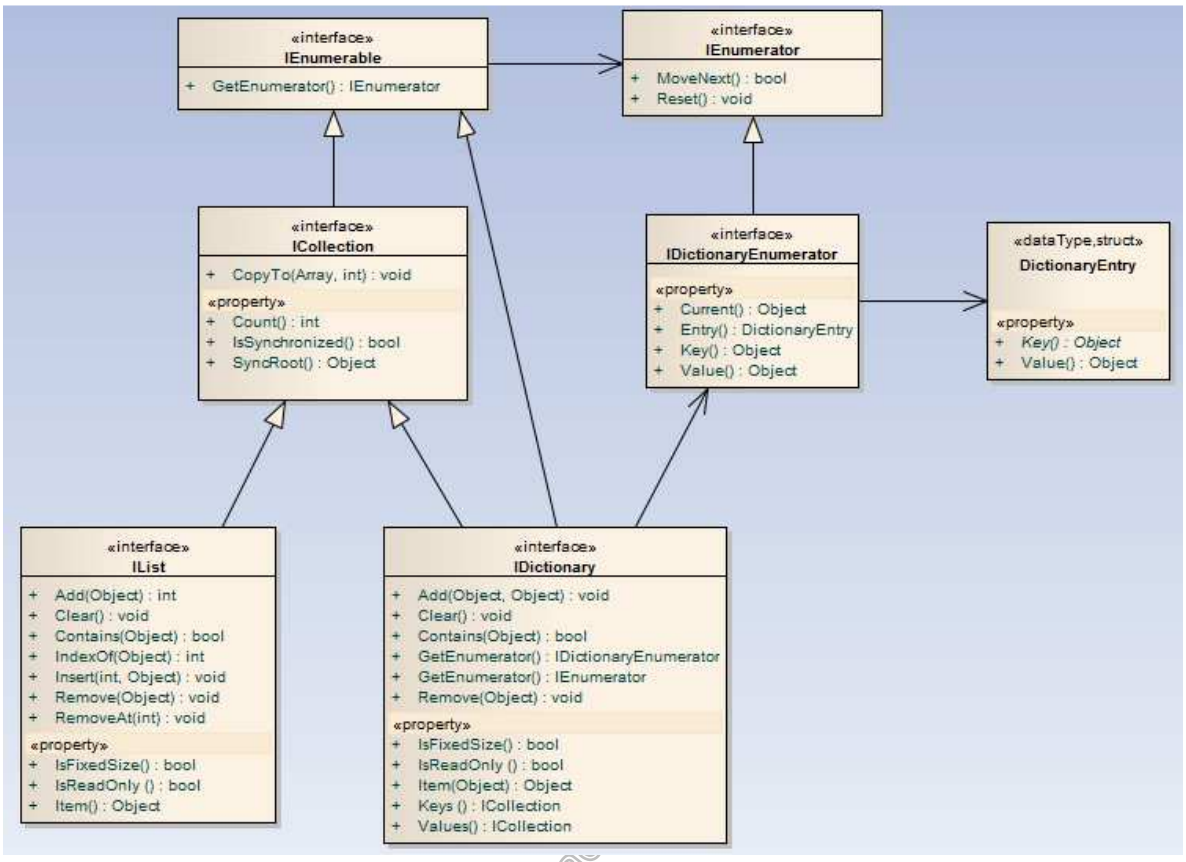
Se pueden utilizar las clases de colección para objetos en cualquier aplicación basada en un lenguaje del Framework .NET.

El espacio de nombres System.Collections define las interfaces que especifican el comportamiento de la mayoría de las colecciones en el Framework .NET. Se pueden organizar las interfaces en tres grupos, a saber:

- Interfaces de colecciones
- Interfaces de comparaciones
- Interfaces de enumeraciones

Interfaces de colecciones

El espacio de nombres System.Collections define cuatro interfaces de la colección (que utilizan otras dos interfaces y una estructura en las definiciones de propiedades y métodos que contienen), según muestra el siguiente gráfico:



Las cuatro interfaces principales se describen a continuación:

- **IEnumerable**

  Define el método GetEnumerator, que devuelve un objeto que implementa la interfaz IEnumerator. Se puede utilizar el enumerador para recorrer en iteración los elementos de una colección basada en objetos.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
GetEnumerator	**IEnumerator**	El objeto del tipo IEnumerator utilizado para recorrer la colección	**N / A**	**N / A**	Retorna un enumerador que sirve para recorrer una colección.

- **ICollection**

  Hereda de la interfaz IEnumerable y define el tamaño, los enumeradores y los miembros de sincronización que se pueden utilizar con todas las colecciones basadas en objetos. En la tabla siguiente se describen las propiedades públicas de la interfaz ICollection.

Propiedad	Descripción
Count	Obtiene el número de elementos que están contenidos en objeto del tipo ICollection.
IsSynchronized	Devuelve un valor booleano que indica si el acceso al objeto ICollection está sincronizado. Si el objeto ICollection está sincronizado, es seguro acceder a sus elementos desde hilos (threads) separados.
SyncRoot	Obtiene un objeto que se puede utilizar para sincronizar el acceso al objeto ICollection. Se puede utilizar el objeto de que este método retorna con una instrucción lock o SyncLock (C# o VB) para un asegurar acceso thread-safe (que los threads no corrompan datos) a los elementos dentro el objeto ICollection.

La interfaz ICollection define un método CopyTo que copia los elementos de la interfaz ICollection en un vector, comenzando en un índice determinado del mismo.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
CopyTo	**void**	**N / A**	**Array**	Un objeto del tipo Array de una dimensión el cual es el destino de los elementos copiados del objeto del tipo Ilist. El índice debe comenzar en 0	Copia los elementos de un objeto del tipo ICollection en un objeto del tipo Array, comenzando a partir del índice indicado
			int o Integer	Un índice en base a 0 en el vector donde comienza la copia	

Las excepciones que puede lanzar el método en las clases que implementen la interfaz son:

Método	Excepción	Descripción
CopyTo	**ArgumentNullException**	El objeto del tipo Array es nulo
	ArgumentOutOfRangeException	El índice es menor a 0
	ArgumentException	El objeto del tipo Array es multidimensiopnal o el número de elementos en el objeto del tipo ICollection es más grande que el espacio desde el indice provisto hasta el fin del objeto de tipo Array o no se puede convertir automáticamente el objeto del tipo ICollection al objeto del tipo Array

- **IList**

  Hereda de la interfaz ICollection y representa una colección de objetos que se puede acceder

individualmente por índice. En la siguiente tabla se describen las propiedades públicas de la interfaz IList.

Propiedad	Descripción
IsFixedSize	Retorna un valor booleano que indica si el objeto del tipo IList tiene un tamaño fijo.
IsReadOnly	Retorna un valor booleano que indica si el objeto del tipo IList es de sólo lectura.
Item	Obtiene o establece el elemento especificado en el índice a partir del cero del objeto del tipo IList. Se puede utilizar la notación NombreX[índice] o NombreX(índice) (C# o VB) como indexador para acceder a un elemento en el índice especificado.

La siguiente tabla describe los métodos públicos de los objetos del tipo IList.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Add	**int**	La posición donde se insertó el nuevo elemento. Retorna -1 si no se inserta en la colección	**Array**	Un objeto del tipo Array de una dimensión el cual es el destino de los elementos copiados del objeto del tipo IList. El indice debe comenzar en 0	Agrega un elemento al objeto del tipo IList.
			int o Integer	Un índice en base a 0 en el vector donde comienza la copia	
Clear	**void**	N / A	**N / A**	N / A	Remueve todos los objetos contenidos dentro de un objeto del tipo IList
Contains	**bool**	Verdadero (true o True) si se encuentra el objeto en la colección	**Object**	El objeto a localizar dentro del objeto del tipo IList	Determina si el objeto del tipo IList contiene un artículo específico.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
IndexOf	**int**	La posición donde se encontró el objeto. Retorna -1 en caso contrario	**Object**	El objeto a localizar dentro del objeto del tipo IList	Retorna el índice de un objeto específico en elobjeto del tipo IList, o devuelve -1 si el objeto del tipo IList no contiene ningún objeto.
Insert	**void**	N / A	**int o Integer**	Un índice en base a 0 en el objeto del tipo IList donde se insertará el objeto recibido como argumento	Inserta un elemento en el objeto del tipo IList en el índice especificado.
			Object	El objeto como argumento a insertar en el objeto del tipo IList	
Remove	**void**	N / A	**Object**	El objeto como argumento a remover en el objeto del tipo IList	Quita la primera aparición de un objeto específico del objeto del tipo IList.
RemoveAt	**void**	N / A	**int o Integer**	Un índice en base a 0 en el objeto del tipo IList donde se borrará el objeto	Elimina el elemento en el índice especificado del objeto del tipo IList.

Las excepciones que pueden lanzar los métodos que implementen la interfaz son:

Método	Excepción	Descripción
Add	**NotSupportedException**	El objeto del tipo IList es de sólo lectura o tiene un tamaño fijo
Clear	**NotSupportedException**	El objeto del tipo IList es de sólo lectura
Contains	**N / A**	N / A
IndexOf	**N / A**	N / A
Insert	**ArgumentOutOfRangeException**	El índice no es válido en el objeto del tipo IList
	NotSupportedException	El objeto del tipo IList es de sólo lectura o tiene un tamaño fijo
	NullReferenceException	El objeto a insertar es una referencia nula en el objeto del tipo IList

Método	Excepción	Descripción
Remove	**NotSupportedException**	El objeto del tipo IList es de sólo lectura o tiene un tamaño fijo
RemoveAt	**ArgumentOutOfRangeException**	El índice no es válido en el objeto del tipo IList
	NotSupportedException	El objeto del tipo IList es de sólo lectura o tiene un tamaño fijo

- **IDictionary**

  Hereda de las interfaces ICollection e IEnumerable y representa una colección de objetos en la forma de pares clave / valor. En la siguiente tabla se describen las propiedades públicas de la interfaz IDictionary.

Propiedad	Descripción
IsFixedSize	Retorna un valor booleano que indica si el objeto del tipo IDictionary tiene un tamaño fijo.
IsReadOnly	Retorna un valor booleano que indica si el objeto del tipo IDictionary es de sólo lectura.
Item	Obtiene o establece el elemento especificado en el índice a partir del cero del objeto del tipo IDictionary. Se puede utilizar la notación NombreX[índice] o NombreX(índice) (C# o VB) como indexador para acceder a un elemento en el índice especificado.
Keys	Obtiene un objeto del tipo ICollection que contiene las claves del objeto del tipo IDictionary
Values	Obtiene un objeto del tipo ICollection que contiene los valores del objeto del tipo IDictionary

La siguiente tabla describe los métodos públicos de los objetos del tipo IDictionary.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Add	**void**	N / A	**Object**	Un objeto que se utiliza como clave de la entrada al objeto del tipo IDictionary que se está agregando	Agrega un elemento al objeto del tipo IDictionary que se considera un entrada compuesta por un par de

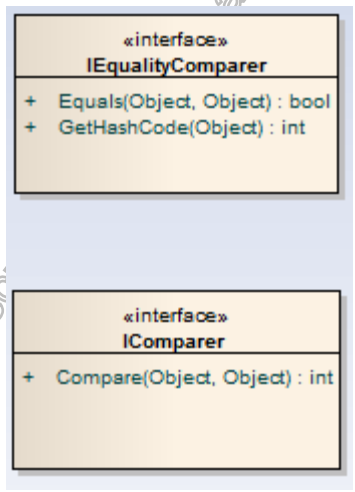
Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
			Object	Un objeto que se utiliza como valor de la entrada al objeto del tipo IDictionary que se está agregando	objetos con formato clave / valor.
Clear	**void**	N / A	**N / A**	N / A	Remueve todos los objetos contenidos dentro de un objeto del tipo IDictionary
Contains	**bool**	Verdadero (true o True) si se encuentra el objeto clave en la colección	**Object**	El objeto clave a localizar dentro del objeto del tipo IDictionary	Determina si el objeto del tipo IDictionary contiene un artículo específico.
Remove	**void**	N / A	**Object**	El objeto clave como argumento a remover en el objeto del tipo IDictionary	Quita un objeto con la clave específicada del objeto del tipo IDictionary.
GetEnumerator	**IDictionaryEnumerator**	El enumerador de un objeto de tipo IDictionaryEnumerator para un objeto del tipo IDictionary	**N / A**	N / A	Permite acceder a claves, valores y entradas (pares clave / valor) que el enumerador posee como propiedades

Las excepciones que pueden lanzar los métodos que implementen la interfaz son:

Método	Excepción	Descripción
Add	**ArgumentNullException**	El objeto clave es una referencia nula
	ArgumentException	Existe un elemento con la misma clave en el objeto del tipo IDictionary
	NotSupportedException	El objeto del tipo IDictionary es de sólo lectura o tiene un tamaño fijo
Clear	**NotSupportedException**	El objeto del tipo IDictionary es de sólo lectura
Contains	**ArgumentNullException**	El objeto clave es una referencia nula
GetEnumerator	**N / A**	N / A
Remove	**NotSupportedException**	El objeto del tipo IDictionary es de sólo lectura o tiene un tamaño fijo
	ArgumentNullException	El objeto clave es una referencia nula

Interfaces de comparación

El espacio de nombres System.Collections define dos interfaces de comparación, como muestra el siguiente gráfico:



- **IComparer**

  Define un método Compare que compara dos objetos y devuelve un valor para indicar si uno es menor que, igual a, o mayor que el otro. La implementación define lo que significa para dos objetos ser menor que, igual a, o mayor que el otro. Varias clases de colección proporcionan un constructor que toma un parámetro IComparer, que le permite especificar qué implementación de IComparer se utilizará para dicha colección. La implementación predeterminada de IComparer es la clase Comparer, que realiza comparaciones entre cadenas diferenciando mayúsculas y minúsculas (sensibles al caso).

Si se desea realizar comparaciones de cadenas que no sean sensibles al caso, utilizar la clase CaseInsensitiveComparer.

La siguiente tabla muestra el método de la interfaz.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Compare	**int**	Si es menor a cero el primer argumento es menor que el segundo. Si es igual a cero el primer argumento es igual que el segundo. Si es mayor a cero el primer argumento es mayor que el segundo.	**Object**	Primer objeto a comparar	Compara dos objetos y retorna el resultado
			Object	Segundo objeto a comparar	

La excepción que puede lanzar una clase que implemente la interfaz es:

Método	Excepción	Descripción
Compare	**ArgumentException**	Ni el primer ni el segundo objeto implementan la interfaz IComparable o los objetos son de distinto tipo y no se puede manejar la comparación

- **IEqualityComparer**

  Define un método Equals que determina si los objetos especificados son iguales. La interfaz también define un método GetHashCode que devuelve un código hash para el dicho objeto. La siguiente tabla muestra los métodos de la interfaz.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Equals	**bool**	Retorna ture o True si ambos objetos son iguales	**Object**	Primer objeto a comparar	Determina si dos objetos son iguales
			Object	Segundo objeto a comparar	
GetHashCode	**int**	El valor del código hash de un objeto específico	**Object**	El objeto respecto del cual se va a retornar el código hash	Retorna el código hash de un objeto específico

Las excepciones que puede lanzar una clase que implemente la interfaz son:

Método	Excepción	Descripción
Equals	**ArgumentException**	Los objetos son de distinto tipo y / o no se puede manejar la comparación
GetHashCode	**ArgumentNullException**	La referencia recibida como argumento es nula

El espacio de nombres System define una interfaz que permite a un objeto compararse a sí mismo contra otro objeto, como la siguiente:

- **IComparable**
  Define un método CompareTo que compara el objeto actual contra otro objeto y devuelve un valor para indicar si el objeto actual es menor que, igual a, o mayor que el otro objeto. La interfaz IComparable es implementada por los tipos cuyos valores se pueden ordenar, como las clases de cadenas y numéricas. Se puede implementar IComparable en clases y estructuras propias para crear un método de comparación específico del tipo que sea adecuado para fines tales como la clasificación u ordenamiento. La siguiente tabla muestra el método de la interfaz.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
CompareTo	**int**	Si es menor a cero la instancia actual es menor que el argumento. Si es igual a cero la instancia actual es igual que el argumento. Si es mayor a cero la instancia actual es mayor que el argumento.	**Object**	El objeto a comparar con la instacia actual	Compara la instancia actual del objeto con otra instancia del mismo tipo y retorna el resultado

La excepción que puede lanzar una clase que implemente la interfaz es:

Método	Excepción	Descripción
CompareTo	**ArgumentException**	El objeto recibido es de distinto tipo que la instacia actual.

Para comprender como se debe utilizar la interfaz se presenta el siguiente ejemplo, cuyas clases serán utilizadas posteriormente como objetos que se almacenen dentro de distintas colecciones. Como las clases que se muestran a continuación son reutilizadas, lo adecuado es colocarlas dentro de una librería de clases a la cual se referencie desde diferentes proyectos (es más, este es el caso para los ejemplos presentados ya que las clases a continuación se presentan el el formato no genérico en la librería de clases ***estudiantes*** y en la librería con la implementación genérica llamada ***estudiantesT***). Notar que es buena práctica rescribir los métodos Equals y GetHashCode de Object a pesar de implementar la interfaz. Esto asegura la mayoría de los llamados posibles para comparaciones que se puedan realizar.

Ejemplo

C#

```
namespace estudiantes
{
    public class Alumno : IComparable
    {
        private string apellido;
        private string nombre;
        private string segundoNombre;
        private int edad;
        private int legajo;

        public Alumno(string nombre, string segundoNombre,
              string apellido, int edad, int legajo)
        {
            this.apellido = apellido;
            this.nombre = nombre;
            this.segundoNombre = segundoNombre;
            this.edad = edad;
            this.legajo = legajo;
        }

        public int CompareTo(object obj)
        {
            if (obj == null) return 1;
            Alumno elOtro = obj as Alumno;
            if (elOtro != null)
            {
               int aux = apellido.CompareTo(elOtro.apellido);
               if (aux != 0) return aux;
               aux = nombre.CompareTo(elOtro.nombre);
               if (aux != 0) return aux;
               aux = segundoNombre.CompareTo(elOtro.segundoNombre);
               if (aux != 0) return aux;
               return 0;
            }
            else
               throw new ArgumentException("El objeto no es un alumno");
        }
```

```
// Rescribir GestHasCode de Object para las
// colecciones que ordenan por este método
public override int GetHashCode()
{
    const int primo = 31;
    int resultado = 1;
    resultado = primo * resultado
            + ((apellido == null) ? 0 : apellido.GetHashCode());
    resultado = primo * resultado +
             ((nombre == null) ? 0 : nombre.GetHashCode());
    resultado = primo * resultado +
             ((segundoNombre == null) ? 0 : segundoNombre.GetHashCode());

    return resultado;
}

// Rescribir el método Equals de Object
// y cumplir con las relaciones de equivalencia
public override bool Equals(Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (GetType() != obj.GetType())
        return false;
    Alumno otro = (Alumno)obj;
    if (apellido == null)
    {
        if (otro.apellido != null)
            return false;
    }
    else if (!apellido.Equals(otro.apellido))
        return false;
    if (nombre == null)
    {
        if (otro.nombre != null)
            return false;
    }
    else if (!nombre.Equals(otro.nombre))
        return false;
    if (segundoNombre == null)
    {
       if (otro.segundoNombre != null)
            return false;
    }
    else if (!segundoNombre.Equals(otro.nombre))
        return false;

    return true;
}

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

```
}

public string Apellido
{
    get
    {
        return apellido;
    }
    set
    {
        apellido = value;
    }
}
public string Nombre
{
    get
    {
        return nombre;
    }
    set
    {
        nombre = value;
    }
}
public string SegundoNombre
{
    get
    {
        return segundoNombre;
    }
    set
    {
        segundoNombre = value;
    }
}
public int Edad
{
    get
    {
        return edad;
    }
    set
    {
        edad = value;
    }
}
public int Legajo
{
    get
    {
        return legajo;
    }
    set
    {
        legajo = value;
```

```
            }
        }
    }
}

namespace estudiantes
{
    public class Horario: IComparable
    {
        private int dia = 0;
        private int horaComienzo = 0;
        private int minutosComienzo = 0;
        private int horaFin = 0;
        private int minutosFin = 0;

        public Horario(int dia, int horaComienzo, int minutosComienzo, int horaFin,
                int minutosFin)
        {
            this.dia = dia;
            this.horaComienzo = horaComienzo;
            this.minutosComienzo = minutosComienzo;
            this.horaFin = horaFin;
            this.minutosFin = minutosFin;
        }

        public Horario(Horario f)
        {
            this.dia = f.dia;
            this.horaComienzo = f.horaComienzo;
            this.minutosComienzo = f.minutosComienzo;
            this.horaFin = f.horaFin;
            this.minutosFin = f.minutosFin;
        }

        public Horario Agregar(int masDias)
        {
            Horario nuevaFecha = new Horario(this);
            nuevaFecha.dia += masDias;
            return nuevaFecha;
        }

        public void Imprimir()
        {
            Console.WriteLine("Horario--> ");
            Console.WriteLine("Día: " + dia);
            Console.WriteLine("Hora de comienzo: " + horaComienzo);
            Console.WriteLine("Minutos de comienzo: " + minutosComienzo);
            Console.WriteLine("Hora de fin: " + horaFin);
            Console.WriteLine("Minutos de fin: " + minutosFin);
        }

        // Rescribir GestHasCode de Object para las
        // colecciones que ordenan por este método
        public override int GetHashCode()
        {
```

```
        return dia ^ horaComienzo ^ minutosComienzo ^ horaFin ^ minutosFin;
    }

    // Rescribir el método Equals de Object
    // y cumplir con las relaciones de equivalencia
    public override bool Equals(Object obj)
    {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (GetType() != obj.GetType())
            return false;
        Horario otro = (Horario)obj;
        if (dia != otro.dia) return false;
        if (horaComienzo != otro.horaComienzo) return false;
        if (minutosComienzo != otro.minutosComienzo) return false;
        if (horaFin != otro.horaFin) return false;
        if (minutosFin != otro.minutosFin) return false;
        return true;
    }

    public override string ToString()
    {
        return "[" + dia + " " + horaComienzo + ":" + minutosComienzo +
            "] a [" + dia + " " + horaFin + ":" + minutosFin + "]";
    }

    public int Dia
    {
        get
        {
            return dia;
        }
    }
    public int HoraComienzo
    {
        get
        {
            return horaComienzo;
        }
    }
    public int MinutosComienzo
    {
        get
        {
            return minutosComienzo;
        }
    }
    public int HoraFin
    {
        get
        {
            return horaFin;
        }
```

```
        }
        public int MinutosFin
        {
            get
            {
                return minutosFin;
            }
        }

        public int CompareTo(object obj)
        {
            if (obj == null) return 1;
            Horario elOtro = obj as Horario;
            if (elOtro != null)
            {
                int aux = dia - elOtro.dia;
                if (aux != 0) return aux;
                aux = horaComienzo - elOtro.horaComienzo;
                if (aux != 0) return aux;
                aux = horaFin - elOtro.horaFin;
                if (aux != 0) return aux;
                aux = minutosComienzo - elOtro.minutosComienzo;
                if (aux != 0) return aux;
                aux = minutosFin - elOtro.minutosFin;
                if (aux != 0) return aux;
                return 0;
            }
            else
                throw new ArgumentException("El objeto no es un horario");
        }
    }
}
```

VB

```
Public Class Alumno
    Implements IComparable

    Private _apellido As String
    Private _nombre As String
    Private _segundoNombre As String
    Private _edad As Integer
    Private _legajo As Integer

    Public Sub New(_nombre As String, _segundoNombre As String, _
       _apellido As String, _edad As Integer, _legajo As Integer)
        Me._apellido = _apellido
        Me._nombre = _nombre
        Me._segundoNombre = _segundoNombre
        Me._edad = _edad
        Me._legajo = _legajo
    End Sub

    Public Function CompareTo(obj As Object) As Integer Implements _
       System.IComparable.CompareTo
        If obj Is Nothing Then
```

```
            Return 1
        End If
        Dim elOtro As Alumno = TryCast(obj, Alumno)
        If Not elOtro Is Nothing Then
            Dim aux As Integer = _apellido.CompareTo(elOtro._apellido)
            If aux <> 0 Then Return aux
            aux = _nombre.CompareTo(elOtro._nombre)
            If aux <> 0 Then Return aux
            aux = _segundoNombre.CompareTo(elOtro._segundoNombre)
            If aux <> 0 Then Return aux
            Return 0
        Else
            Throw New ArgumentException("El objeto no es un alumno")
        End If
    End Function

    ' Rescribir GestHasCode de Object para las
    ' colecciones que ordenan por este método
    Public Overrides Function GetHashCode() As Integer
        Const primo As Integer = 31
        Dim resultado As Integer = 1
        resultado = primo * resultado + _
            IIf((_apellido Is Nothing), 0, _apellido.GetHashCode())
        resultado = primo * resultado + _
            IIf((_nombre Is Nothing), 0, _nombre.GetHashCode())
        resultado = primo * resultado + _
            IIf((_segundoNombre Is Nothing), 0, _segundoNombre.GetHashCode())
        Return resultado
    End Function

    ' Rescribir el método Equals de Object
    ' y cumplir con las relaciones de equivalencia
    Public Overrides Function Equals(obj As Object) As Boolean
        If Me Is obj Then
            Return True
        End If

        If obj Is Nothing Then
            Return False
        End If
        If Me.GetType() <> obj.GetType() Then
            Return False
        End If

        Dim otro As Alumno = CType(obj, Alumno)

        If _nombre Is Nothing Then
            If Not otro._nombre Is Nothing Then
                Return False
            End If
        ElseIf Not _nombre.Equals(otro._nombre) Then
            Return False
        End If

        If _segundoNombre Is Nothing Then
```

```
            If Not otro._segundoNombre Is Nothing Then
                Return False
            End If
        ElseIf Not _segundoNombre.Equals(otro._segundoNombre) Then
            Return False
        End If

        If _apellido Is Nothing Then
            If Not otro._apellido Is Nothing Then
                Return False
            End If
        ElseIf Not _apellido.Equals(otro._apellido) Then
            Return False
        End If

        Return True
    End Function

    Public Overrides Function ToString() As String
        Return "[" + Apellido + ", " + Nombre + " " + SegundoNombre + "]"
    End Function

    Public Property Apellido() As String
        Get
            Return _apellido
        End Get
        Set(ByVal value As String)
            _apellido = value
        End Set
    End Property
    Public Property Nombre() As String
        Get
            Return _nombre
        End Get
        Set(ByVal value As String)
            _nombre = value
        End Set
    End Property
    Public Property SegundoNombre() As String
        Get
            Return _segundoNombre
        End Get
        Set(ByVal value As String)
            _segundoNombre = value
        End Set
    End Property
    Public Property Edad() As Integer
        Get
            Return _edad
        End Get
        Set(ByVal value As Integer)
            _edad = value
        End Set
    End Property
    Public Property Legajo() As Integer
```

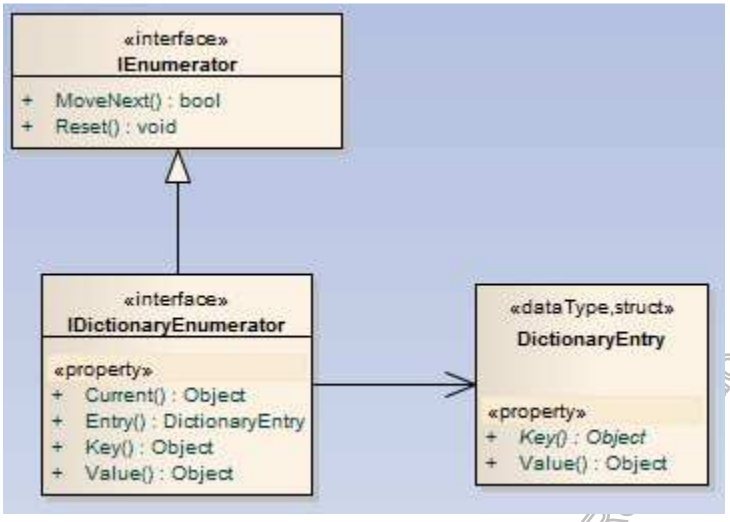
```
        Get
            Return _legajo
        End Get
        Set(ByVal value As Integer)
            _legajo = value
        End Set
    End Property
End Class

Public Class Horario
    Implements IComparable

    Private _dia As Integer = 0
    Private _horaComienzo As Integer = 0
    Private _minutosComienzo As Integer = 0
    Private _horaFin As Integer = 0
    Private _minutosFin As Integer = 0

    Public Sub New(ByVal _dia As Integer, ByVal _horaComienzo As Integer, _
                ByVal _minutosComienzo As Integer, _
                ByVal _horaFin As Integer, ByVal _minutosFin As Integer)
        Me._dia = _dia
        Me._horaComienzo = _horaComienzo
        Me._minutosComienzo = _minutosComienzo
        Me._horaFin = _horaFin
        Me._minutosFin = _minutosFin
    End Sub

    Public Sub New(ByVal f As Horario)
        Me._dia = f._dia
        Me._horaComienzo = f._horaComienzo
        Me._minutosComienzo = f._minutosComienzo
        Me._horaFin = f._horaFin
        Me._minutosFin = f._minutosFin
    End Sub

    Public Function Agregar(ByVal masDias As Integer) As Horario
        Dim nuevaFecha As New Horario(Me)
        nuevaFecha._dia += masDias
        Return nuevaFecha
    End Function

    Public Sub Imprimir()
        Console.WriteLine("Horario: ")
        Console.WriteLine("Día: " + _dia.ToString)
        Console.WriteLine("Hora de comienzo: " + _horaComienzo.ToString)
        Console.WriteLine("Minutos de comienzo: " + _minutosComienzo.ToString)
        Console.WriteLine("Hora de fin: " + _horaFin.ToString)
        Console.WriteLine("Minutos de fin: " + _minutosFin.ToString)
    End Sub

    Public Overrides Function GetHashCode() As Integer
        Return _dia Xor _horaComienzo Xor _
             _minutosComienzo Xor _horaFin Xor _minutosFin
    End Function
```

```
Public Overrides Function Equals(obj As Object) As Boolean
    If Me Is obj Then
        Return True
    End If
    If obj Is Nothing Then
        Return False
    End If
    If Me.GetType() <> obj.GetType() Then
        Return False
    End If

    Dim otro As Horario = CType(obj, Horario)

    If _dia <> otro.Dia Then Return False
    If _horaComienzo <> otro._horaComienzo Then Return False
    If _minutosComienzo <> otro._minutosComienzo Then Return False
    If _horaFin <> otro._horaFin Then Return False
    If _minutosFin <> otro._minutosFin Then Return False
    Return True
End Function

Public Overrides Function ToString() As String
    Return "[" + _dia.ToString + " " + _
          _horaComienzo.ToString + ":" + _minutosComienzo.ToString + _
           "] a [" + _dia.ToString + " " + _
          _horaFin.ToString + ":" + _minutosFin.ToString + "]"
End Function

Public Function CompareTo(obj As Object) As Integer _
   Implements System.IComparable.CompareTo
    If obj Is Nothing Then
        Return 1
    End If
    Dim elOtro As Horario = TryCast(obj, Horario)
    If Not elOtro Is Nothing Then
        Dim aux As Integer = _dia - elOtro._dia
        If aux <> 0 Then Return aux
        aux = _horaComienzo - elOtro._horaComienzo
        If aux <> 0 Then Return aux
        aux = _horaFin - elOtro._horaFin
        If aux <> 0 Then Return aux
        aux = _minutosComienzo - elOtro._minutosComienzo
        If aux <> 0 Then Return aux
        aux = _minutosFin - elOtro._minutosFin
        If aux <> 0 Then Return aux
        Return 0
    Else
        Throw New ArgumentException("El objeto no es un alumno")
    End If
End Function
Public Property Dia() As Integer
    Get
        Return _dia
    End Get
```

```
        Set(ByVal value As Integer)
            _dia = value
        End Set
    End Property
    Public Property HoraComienzo() As Integer
        Get
            Return _horaComienzo
        End Get
        Set(ByVal value As Integer)
            _horaComienzo = value
        End Set
    End Property
    Public Property MinutosComienzo() As Integer
        Get
            Return _minutosComienzo
        End Get
        Set(ByVal value As Integer)
            _minutosComienzo = value
        End Set
    End Property
    Public Property HoraFin() As Integer
        Get
            Return _horaFin
        End Get
        Set(ByVal value As Integer)
            _horaFin = value
        End Set
    End Property
    Public Property MinutosFin() As Integer
        Get
            Return _minutosFin
        End Get
        Set(ByVal value As Integer)
            _minutosFin = value
        End Set
    End Property
End Class
```


Interfaces de enumeración

El espacio de nombres System.Collections define dos interfaces de enumeración, como muestra la siguiente figura:



- **IEnumerator**

  La interfaz se utiliza para soportar la enumeración de objetos en una colección. Se puede obtener una interfaz IEnumerator al invocar el método GetEnumerator que pertenezca a un objeto del tipo ICollection. IEnumerator define una propiedad que permite interactuar con el elemento actual de la colección, un método MoveNext que hace avanzar el enumerador al siguiente elemento de la colección y un método Reset establece el enumerador en su posición inicial. En la siguiente tabla se describe la propiedad pública de la interfaz.

Propiedad	Descripción
Current	Obtiene el elemento actual que está contenido en objeto del tipo ICollection.

La siguiente tabla muestra los métodos de la interfaz.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
MoveNext	**bool**	Retorna verdadero si el enumerador avanza exitósamente al próximo elemento y falso en caso contrario	**N / A**	**N / A**	Avanza el enumerador al próximo elemento de la colección
Reset	**void**	N / A	**N / A**	**N / A**	Coloca al enumerador en la posición inicial, la cual es antes del primer elemento de la colección

Las excepciones que puede lanzar una clase que implemente la interfaz son:

Método	Excepción	Descripción
MoveNext	**InvalidOperationException**	La colección fue modificada luego de haber creado el enumerador
Reset	**InvalidOperationException**	La colección fue modificada luego de haber creado el enumerador

- **IDictionaryEnumerator**

  La interfaz se utiliza para soportar la enumeración de objetos en una colección y hereda de la interfaz IEnumerator para crear una enumeración de objetos del tipo diccionario que constan de una estructura de datos que incluyen el objeto utilizado como clave y el utilizado como valor. Se obtiene un objeto del tipo IDictionaryEnumerator al invocar al método GetEnumerator en un objeto del tipo IDictionary. Además de los miembros que se definen en la interfaz base IEnumerator, se definen IDictionaryEnumerator Entry, Key y Value que le permiten acceder a la entrada actual (del tipo entrada de diccionario), la clave y el valor de la entrada en la cual está posicionado el enumerador dentro del diccionario. En la siguiente tabla se describen las propiedades públicas de la interfaz IDictionaryEnumerator sin incluir la propiedad Current heredada de IEnumerator porque hace lo mismo que Entry y esta última es la que se utiliza generalmente (esto simplemente responde a que es más descriptiva).

Propiedad	Descripción
Entry	Obtiene el elemento (DictionaryEntry) actual que está en la colección.
Key	Obtiene un objeto que contiene la clave de la enumeración.
Value	Obtiene un objeto que contiene el valor de la enumeración.

La siguiente tabla muestra los métodos de la interfaz.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
MoveNext	**bool**	Retorna verdadero si el enumerador avanza exitósamente al próximo elemento y falso en caso contrario	**N / A**	**N / A**	Avanza el enumerador al próximo elemento de la colección
Reset	**void**	N / A	**N / A**	**N / A**	Coloca al enumerador en la posición inicial, la cual es antes del primer elemento de la colección

Las excepciones que puede lanzar una clase que implemente la interfaz son:

Método	Excepción	Descripción
MoveNext	**InvalidOperationException**	La colección fue modificada luego de haber creado el enumerador
Reset	**InvalidOperationException**	La colección fue modificada luego de haber creado el enumerador

El siguiente ejemplo muestra como implementar una interfaz IDictionary, lo cual por herencia demanda la implementación de las interfaces IEnumerable e ICollection. Además debe gestionar tanto un IEnumerator como un IDictionaryEnumerator para que este bien definida la colección y se pueda recorrer.

Como los diccionarios son pares clave / valor se utiliza IDictionaryEnumerator para recorrerlo. Según como se implemente el método Add de la colección, se puede llamar tanto a Equals como a ComparteTo para realizar el agregado de un objeto.

Ejemplo

C#

```
namespace idiccionario
{
    class Diccionario : IDictionary
    {
        // El vector de elementos
        private DictionaryEntry[] elementos;
        private Int32 elementosEnUso = 0;

        // Construir el Diccionario con el número deseado de elementos.
        // El número de elementos no se puede cambiar durante
        // la vida útil de este Diccionario.
        public Diccionario(Int32 cantidad)
        {
            elementos = new DictionaryEntry[cantidad];
        }


        #region IDictionary Members
        public bool IsReadOnly { get { return false; } }
        public bool Contains(object key)
        {
            Int32 indice;
            return TratarDeObtenerElIndiceDeLaClave(key, out indice);
        }
        public bool IsFixedSize { get { return false; } }
        public void Remove(object clave)
        {
            if (clave == null) throw new ArgumentNullException(
                    "La clave no puede ser nula");
            // Tratar de encontrar la clave en el vector de DictionaryEntry
            Int32 indice;
            if (TratarDeObtenerElIndiceDeLaClave(clave, out indice))
            {
```

```
            // Si se encuentra la clave, correr todos los elementos.
            Array.Copy(elementos, indice + 1,
                elementos, indice, elementosEnUso - indice - 1);
            elementosEnUso--;
        }
        else
        {
            // Si la clave no se encuentra, retornar sin hacer nada.
        }
    }
    public void Clear() { elementosEnUso = 0; }
    public void Add(object clave, object valor)
    {
        if (elementosEnUso == elementos.Length)
            throw new InvalidOperationException(
                "El diccionario no puede almacenar más valores.");
        if (elementosEnUso > 0)
        {
            foreach (DictionaryEntry elemento in elementos)
            {
                if (elemento.Key != null && elemento.Key.Equals(clave))
                    throw new ArgumentException(
                       "La clave existe: " + elemento.Key.ToString());
            }
        }
        elementos[elementosEnUso++] = new DictionaryEntry(clave, valor);
    }
    public ICollection Keys
    {
        get
        {
            // Retornar un  vector donde cada elemento es una clave
            Object[] claves = new Object[elementosEnUso];
            for (Int32 n = 0; n < elementosEnUso; n++)
                claves[n] = elementos[n].Key;

            return claves;
        }
    }
    public ICollection Values
    {
        get
        {
            // Retornar un  vector donde cada elemento es un valor
            Object[] valores = new Object[elementosEnUso];
            for (Int32 n = 0; n < elementosEnUso; n++)
                valores[n] = elementos[n].Value;
            return valores;
        }
    }
    public object this[object clave]
    {
        get
        {
            // Si la clave está en el diccionario, retornar el valor asociado
```

```
            Int32 indice;
            if (TratarDeObtenerElIndiceDeLaClave(clave, out indice))
            {
                // Se encontró la clave. Retornar el valor.
                return elementos[indice].Value;
            }
            else
            {
                // No se encontró la clave. Retornar null.
                return null;
            }
        }

        set
        {
            // Si la clave está en el diccionario, cambiar el valor asociado
            Int32 indice;
            if (TratarDeObtenerElIndiceDeLaClave(clave, out indice))
            {
                // Se encontró la clave. Cambiar el valor.
                elementos[indice].Value = value;
            }
            else
            {
                // No se encontró la clave. Agregar este par clave / valor.
                Add(clave, value);
            }
        }
    }

    public IDictionaryEnumerator GetEnumerator()
    {
        // Construir y retornar el enumerador.
        return new EnumeradorDiccionario(this);
    }
    #endregion
    private class EnumeradorDiccionario : IDictionaryEnumerator
    {
        // Una copia de los pares clave / valor del objeto del tipo Diccionario
        DictionaryEntry[] elementos;
        Int32 indice = -1;

        public EnumeradorDiccionario(Diccionario d)
        {
            // Realizar una copia de las entradas actuales del diccionario
            elementos = new DictionaryEntry[d.Count];
            Array.Copy(d.elementos, 0, elementos, 0, d.Count);
        }

        // Retornar el elemento actual.
        public Object Current { get { ValidarIndice(); return elementos[indice];
                                } }

        // Retornar la entrada actual del diccionario.
        public DictionaryEntry Entry
```

```
        {
            get { return (DictionaryEntry)Current; }
        }

        // Retornar la clave del elemento actual.
        public Object Key { get { ValidarIndice(); return elementos[indice].Key;
                                } }

        //  Retornar el valor del elemento actual.
        public Object Value { get { ValidarIndice();
                        return elementos[indice].Value; } }

        // Avanzar al próximo elemento.
        public Boolean MoveNext()
        {
            if (indice < elementos.Length - 1) { indice++; return true; }
            return false;
        }

        // Validar el índice de la enumeración y lanzar una excepción
        // si el índice está fuera de rango
        private void ValidarIndice()
        {
            if (indice < 0 || indice >= elementos.Length)
                throw new InvalidOperationException(
                        "El enumerador está antes o después de la colección.");
        }

        // Restablecer el índice para reiniciar la enumeración.
        public void Reset()
        {
            indice = -1;
        }
    }
    private Boolean TratarDeObtenerElIndiceDeLaClave(Object clave,
         out Int32 indice)
    {
        for (indice = 0; indice < elementosEnUso; indice++)
        {
            // Si la clave se encuentra, devuelve true.
            // También retorna el índice por referencia
            if (elementos[indice].Key.Equals(clave)) return true;
        }

        // Clave no encontrada, devuelve false. Ignorar el índice
        return false;
    }

    #region ICollection Members
    public bool IsSynchronized { get { return false; } }
    public object SyncRoot { get { throw new NotImplementedException(); } }
    public int Count { get { return elementosEnUso; } }
    public void CopyTo(Array array, int index) {
                throw new NotImplementedException(); }
    #endregion
```

```
        #region IEnumerable Members
        IEnumerator IEnumerable.GetEnumerator()
        {
            // Construir y retornar el enumerador.
            return ((IDictionary)this).GetEnumerator();
        }
        #endregion
    }
}
```

VB

```
Imports System
Imports System.Collections

Public Class Diccionario
    Implements IDictionary

    ' El vector de elementos
    Dim _elementos() As DictionaryEntry
    Dim _elementosEnUso As Integer = 0

    ' Construir el Diccionario con el número deseado de elementos.
    ' El número de elementos no se puede cambiar durante
    ' la vida útil de este Diccionario.
    Public Sub New(ByVal cantidad As Integer)
        _elementos = New DictionaryEntry(cantidad - 1) {}
    End Sub

    ' IDictionary Members
    Public ReadOnly Property IsReadOnly() As Boolean _
       Implements IDictionary.IsReadOnly
        Get
            Return False
        End Get
    End Property

    Public Function Contains(ByVal key As Object) As Boolean _
       Implements IDictionary.Contains
        Dim indice As Integer
        Return TratarDeObtenerElIndiceDeLaClave(key, indice)
    End Function

    Public ReadOnly Property IsFixedSize() As Boolean _
       Implements IDictionary.IsFixedSize
        Get
            Return False
        End Get
    End Property

    Public Sub Remove(ByVal clave As Object) Implements IDictionary.Remove
        If clave Is Nothing Then
            Throw New ArgumentNullException("La clave no puede ser nula")
        End If
        ' Tratar de encontrar la clave en el vector de DictionaryEntry
```

```
        Dim indice As Integer
        If TratarDeObtenerElIndiceDeLaClave(clave, indice) Then

            ' Si se encuentra la clave, correr todos los elementos.
            Array.Copy(_elementos, indice + 1, _
              _elementos, indice, (_elementosEnUso - indice) - 1)
            _elementosEnUso = _elementosEnUso - 1
        Else

            ' Si la clave no se encuentra, retornar sin hacer nada.
        End If
    End Sub

    Public Sub Clear() Implements IDictionary.Clear
        _elementosEnUso = 0
    End Sub

    Public Sub Add(ByVal clave As Object, ByVal valor As Object) Implements
IDictionary.Add

        If _elementosEnUso = _elementos.Length Then
            Throw New InvalidOperationException( _
              "El diccionario no puede almacenar más valores.")
        End If
        If _elementosEnUso > 0 Then
            For Each elemento As DictionaryEntry In _elementos
                If Not (elemento.Key Is Nothing) AndAlso _
                    elemento.Key.Equals(clave) Then
                    Throw New ArgumentException("La clave existe: " _
                           + elemento.Key.ToString())
                End If
            Next
        End If
        _elementos(_elementosEnUso) = New DictionaryEntry(clave, valor)
        _elementosEnUso = _elementosEnUso + 1
    End Sub

    Public ReadOnly Property Keys() As ICollection Implements IDictionary.Keys
        Get
            'Retorna un vector en donde cada elemento es una clave.
            'Nota: Declarar claves() para tener un tamaño de _elementosEnUso - 1
            '   asegura que el vector es de tamaño adecuado. En VB.NET
            '   declarar un vector de tamaño N lo crea desde 0
            '   a N elementos, incluyendo N, en lugar de N - 1
            '   que es el comportamiento por defecto en C# y C++.
            Dim claves() As Object = New Object(_elementosEnUso - 1) {}
            Dim n As Integer
            For n = 0 To _elementosEnUso - 1
                claves(n) = _elementos(n).Key
            Next n

            Return claves
        End Get
    End Property
```

```
Public ReadOnly Property Values() As ICollection Implements IDictionary.Values
    Get
        ' Retornar un  vector donde cada elemento es un valor.
        Dim valores() As Object = New Object(_elementosEnUso - 1) {}
        Dim n As Integer
        For n = 0 To _elementosEnUso - 1
            valores(n) = _elementos(n).Value
        Next n

        Return valores
    End Get
End Property

Public Property Item(ByVal clave As Object) As Object _
   Implements IDictionary.Item
    Get
        ' Si la clave está en el diccionario, retornar el valor asociado.
        Dim indice As Integer
        If TratarDeObtenerElIndiceDeLaClave(clave, indice) Then

            ' Se encontró la clave. Retornar el valor.
            Return _elementos(indice).Value
        Else

            ' No se encontró la clave. Retornar Nothing.
            Return Nothing
        End If
    End Get

    Set(ByVal value As Object)
        ' Si la clave está en el diccionario, cambiar el valor asociado.
        Dim indice As Integer
        If TratarDeObtenerElIndiceDeLaClave(clave, indice) Then
            ' Se encontró la clave. Cambiar el valor.
            _elementos(indice).Value = value
        Else

            ' No se encontró la clave. Agregar este par clave / valor.
            Add(clave, value)
        End If
    End Set
End Property

Private Function TratarDeObtenerElIndiceDeLaClave(ByVal clave As Object, _
   ByRef indice As Integer) As Boolean
    For indice = 0 To _elementosEnUso - 1
        ' Si la clave se encuentra, devuelve true.
        ' También retorna el índice por referencia
        If _elementos(indice).Key.Equals(clave) Then
            Return True
        End If
    Next indice

    ' Clave no encontrada, devuelve false. Ignorar el índice.
    Return False
```

```
    End Function

    Private Class EnumeradorDiccionario
        Implements IDictionaryEnumerator

        ' Una copia de los pares clave / valor del objeto del tipo Diccionario
        Dim _elementos() As DictionaryEntry
        Dim _indice As Integer = -1

        Public Sub New(ByVal d As Diccionario)
            ' Realizar una copia de las entradas actuales del diccionario
            _elementos = New DictionaryEntry(d.Count - 1) {}
            Array.Copy(d._elementos, 0, _elementos, 0, d.Count)
        End Sub

        ' Retornar el elemento actual.
        Public ReadOnly Property Current() As Object _
              Implements IDictionaryEnumerator.Current
            Get
                ValidarIndice()
                Return _elementos(_indice)
            End Get
        End Property

        ' Retornar la entrada actual del diccionario.
        Public ReadOnly Property Entry() As DictionaryEntry _
               Implements IDictionaryEnumerator.Entry
            Get
                Return Current
            End Get
        End Property

        ' Retornar la clave del elemento actual.
        Public ReadOnly Property Key() As Object _
              Implements IDictionaryEnumerator.Key
            Get
                ValidarIndice()
                Return _elementos(_indice).Key
            End Get
        End Property

        '  Retornar el valor del elemento actual.
        Public ReadOnly Property Value() As Object _
              Implements IDictionaryEnumerator.Value
            Get
                ValidarIndice()
                Return _elementos(_indice).Value
            End Get
        End Property

        ' Avanzar al próximo elemento.
        Public Function MoveNext() As Boolean _
              Implements IDictionaryEnumerator.MoveNext
            If _indice < _elementos.Length - 1 Then
                _indice = _indice + 1
```

```
            Return True
        End If

        Return False
    End Function

    ' Validar el índice de la enumeración y lanzar una excepción
    ' si el índice está fuera de rango
    Private Sub ValidarIndice()
        If _indice < 0 Or _indice >= _elementos.Length Then
            Throw New InvalidOperationException(
                "El enumerador está antes o después de la colección.")
        End If
    End Sub

    ' Restablecer el índice para reiniciar la enumeración.
    Public Sub Reset() Implements IDictionaryEnumerator.Reset
        _indice = -1
    End Sub

End Class

Public Function GetEnumerator() As IDictionaryEnumerator _
   Implements IDictionary.GetEnumerator

    ' Construir y retornar el enumerador.
    Return New EnumeradorDiccionario(Me)
End Function


' ICollection Members
Public ReadOnly Property IsSynchronized() As Boolean _
   Implements IDictionary.IsSynchronized
    Get
        Return False
    End Get
End Property

Public ReadOnly Property SyncRoot() As Object Implements IDictionary.SyncRoot
    Get
        Throw New NotImplementedException()
    End Get
End Property

Public ReadOnly Property Count() As Integer Implements IDictionary.Count
    Get
        Return _elementosEnUso
    End Get
End Property

Public Sub CopyTo(ByVal array As Array, ByVal index As Integer) _
   Implements IDictionary.CopyTo
    Throw New NotImplementedException()
End Sub
```

```
    ' IEnumerable Members
    Public Function GetEnumerator1() As IEnumerator _
       Implements IEnumerable.GetEnumerator

        '  Construir y retornar el enumerador.
        Return Me.GetEnumerator()
    End Function
End Class
```

Las interfaces IEnumerable e IEnumerator con el ciclo foreach

El ciclo `foreach` o `For Each` (C# o VB) permite extraer uno a uno los elementos de un vector o una colección. Para que una colección pueda ser utilizada con el ciclo foreach o For Each (C# o VB) debe implementar la interfaz IEnumerable, la cual tiene un método, GetEnumerator, que permite que la instrucción recorra la colección a través del enumerador obtenido. Esto además implica, que debe existir un enumerador para la clase que va a ser recorrida con la instrucciómn, por lo tanto para utilizar el ciclo `foreach` o `For Each` (C# o VB) se deben implementar clases con ambas interfaces.

Las interfaces se pueden implementar juntas o por separado para la colección. En el siguiente ejemplo se creo a modo de demostración, una clase anidada para ver la implementación de ambas interfaces por separado, el cual es el caso menos común. Como objeto a contener dentro de la colección se creó una clase simple llamada Persona, como se muestra a continuación.

Ejemplo

C#

```
namespace enumerables
{
    public class Persona
    {
        private string nombre;
        private string apellido;

        public Persona(string nombre, string apellido)
        {
            this.nombre = nombre;
            this.apellido = apellido;
        }
        public string Nombre
        {
            get
            {
                return nombre;
            }
            set
            {
                nombre = value;
            }
        }
        public string Apellido
        {
```

```
            get
            {
                return apellido;
            }
            set
            {
                apellido = value;
            }
        }
    }
}
```

VB

```
Public Class Persona
    Private _nombre As String
    Private _apellido As String

    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
End Class
```


La clase de colección creada para manejar un conjunto de objetos del tipo Persona se llama Publico. La misma tiene implementada la interfaz IEnumerator para manejar el ciclo `foreach` o `For Each` (C# o VB) en una clase anidada llamada EnumeradorPersonas. Notar que la clase anidada es privada. La razón es que no se desea que se cree ninguna instancia de esta clase fuera de la que la contiene porque su funcionalidad es específica, la de manejar el enumerador.

Ejemplo

C#

```
namespace enumerables
{
    class Publico : IEnumerable
```

```
    {
        private static Persona[] personas;

        public Publico(Persona[] _personas)
        {
            personas = new Persona[_personas.Length];
            for (int i = 0; i < _personas.Length; i++)
            {
                personas[i] = _personas[i];
            }
        }

        public IEnumerator GetEnumerator()
        {
           return new EnumeradorPersonas();
        }

        private class EnumeradorPersonas : IEnumerator
        {
            private int indice = -1;

            public object Current
            {
                get
                {
                    try
                    {
                        if (personas[indice] == null)
                           throw new InvalidOperationException("No hay elementos");
                        return personas[indice];
                    }
                    catch (IndexOutOfRangeException)
                    {
                        throw new InvalidOperationException(
                           "No hay más elementos almacenados");
                    }
                }
            }

            public bool MoveNext()
            {
                indice++;
                return (indice < personas.Length);
            }

            public void Reset()
            {
                indice = -1;
            }
        }
    }
}
```

VB

```
Public Class Publico
```

```
    Implements IEnumerable

    Private Shared _personas() As Persona

    Public Sub New(personas() As Persona)
        ReDim _personas(personas.Length - 1)
        For i As Integer = 0 To _personas.Length - 1
            Publico._personas(i) = personas(i)
        Next
    End Sub

    Public Function GetEnumerator() As System.Collections.IEnumerator _
       Implements System.Collections.IEnumerable.GetEnumerator
        Return New EnumeradorPersonas()
    End Function

    Private Class EnumeradorPersonas
        Implements IEnumerator

        Private indice As Integer = -1

        Public ReadOnly Property Current As Object _
              Implements System.Collections.IEnumerator.Current
            Get
                Try
                    If _personas(indice) Is Nothing Then
                        Throw New InvalidOperationException("No hay elementos")
                    End If

                    Return _personas(indice)
                Catch e As IndexOutOfRangeException
                    Throw New InvalidOperationException(
                            "No hay más elementos almacenados")
                End Try
            End Get
        End Property

        Public Function MoveNext() As Boolean _
              Implements System.Collections.IEnumerator.MoveNext
            indice += 1
            Return (indice < _personas.Length)
        End Function

        Public Sub Reset() Implements System.Collections.IEnumerator.Reset
            indice = -1
        End Sub
    End Class

End Class
```

Clases de colecciones

Aunque se pueden hacer colecciones de objetos relacionados mediante vectores, existen algunas limitaciones. El tamaño de un vector siempre es fijo y debe definirse en el momento en el cual se lo programa.

En segundo lugar, un vector sólo puede contener objetos de mismo tipo de datos, el cual es necesario definir en el momento de su instanciación. Además, un vector no impone ningún mecanismo en particular para la inserción y la recuperación de los elementos. Para este propósito, los creadores del Framework de .NET han proporcionado un número de clases para servir como colección de diferentes tipos. Estas clases están presentes en su mayoría en los espacios de nombres System, System.Collections y System.Collections.Generics.

Algunas de las clases más comunes de System.Collections son los siguientes:

Clase	Descripción
ArrayList	Proporciona una colección similar a un vector, pero que crece dinámicamente según el número de elementos que se agregan.
Stack	Es una colección que trabaja con el principio Último Entrado Primero Salido (UEPS), es decir, el último elemento introducido es el primer elemento a quitar de la colección.
Queue	Es una colección que trabaja con el principio Primero Entrado Primero Salido (PEPS), es decir, el primer elemento insertado es el primer elemento eliminado de la colección.
HashTable	Proporciona una colección de pares de clave / valor organizados en función de un código hash que actúa de clave.
SortedList	Proporciona una colección de pares de clave / valor, donde los elementos se ordenan según la clave. La artículos son accesibles tanto por las claves y como por el índice.
BitArray	Proporciona una colección de valores booleanos con los que se puede interactuar como si fueran banderas o indicadores.

ArrayList

La clase System.Collections.ArrayList es similar a los vectores, pero puede almacenar elementos de cualquier tipo de datos. Implementa las interfaces IList, ICollection, IEnumerable e ICloneable. No es necesario especificar el tamaño de la colección cuando se utiliza un ArrayList (como se suele hacer en el caso de vectores simples). El tamaño de la ArrayList crece dinámicamente según cambia el número de elementos que contiene. Para disminuir la capacidad de un objeto ArrayList, se invoca al método TrimToSize. Un ArrayList utiliza un vector interno e inicializa su tamaño con un valor predeterminado denominado Capacity. Como el número de elementos aumenta o disminuye, ArrayList ajusta la capacidad del vector en consecuencia haciendo uno nuevo y copia los valores antiguos.

El tamaño de la clase ArrayList es el número total de elementos que están realmente presentes en ella, lo cual se obtiene por la propiedad Count, mientras que la capacidad (Capacity) es el número de elementos que puede contener ArrayList sin crear instancias de un nuevo vector.

Los constructores que admite la clase se muestran en la siguiente tabla:

Constructor	Descripción
ArrayList()	Inicializa una nueva instancia de la clase ArrayList que está vacía y tiene la capacidad inicial predeterminada.
ArrayList(ICollection)	Inicializa una nueva instancia de la clase ArrayList que contiene elementos copiados de la colección especificada y que tiene la misma capacidad inicial que el número de elementos copiados.
ArrayList(Int32)	Inicializa una nueva instancia de la clase ArrayList que está vacía y tiene la capacidad inicial especificada.

Las propiedades de la clase están determinadas por las interfaces implementadas. Los métodos más importantes se enumeran en la siguiente tabla:

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
AddRange	void	N / A	ICollection	El objeto del tipo ICollection que se agregará. La colección no puede ser nula pero puede tener dentro valores nulos	Agrega los elementos de un objeto ICollection al final del objeto ArrayList.
BinarySearch	int	El índice del valor encontrado o un número negativo	Object	Objeto a localizar. Puede ser nulo	Utiliza un algoritmo de búsqueda binaria para localizar un elemento específico en un objeto del tipo ArrayList ordenado.
FixedSize	IList	Objeto del tipo IList con los elementos	IList	La colección a partir de la cual se crea la nueva lista.	Devuelve una lista de tamaño fijo que le permite modificar los elementos pero no para añadir o eliminarlos.
GetRange	ArrayList	La colección con el subconjunto	int	Índice desde donde comenzar	Devuelve un objeto ArrayList que representa un subconjunto de los

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
		de elementos	int	Cantidad de elementos	elementos del objeto de origen ArrayList.
InsertRange	void	N / A	int	Índice desde donde comenzar	Inserta los elementos de una colección en el objeto ArrayList en el índice especificado.
			ICollection	Colección a insertar	
LastIndexOf	int	El ínidice del valor encontrado o un número negativo	Object	Objeto a localizar. Puede ser nulo	Devuelve el índice de base cero de la última aparición de un valor en el ArrayList.
			int	Índice desde dónde buscar hacia atrás	
ReadOnly	IList	Objeto del tipo IList con los elementos	IList	La colección a partir de la cual se crea la nueva lista.	Devuelve un contenedor de lista de sólo lectura para el objeto ArrayList.
RemoveRange	void	N / A	int	Índice desde donde comenzar	Quita todos los elementos del objeto del tipo IList.
			int	Cantidad de elementos	
Repeat	ArrayList	Objeto del tipo IList con la cantidad de elementos repetidos indicado	Object	Objeto a copiar múltiples veces. Puede ser nulo	Devuelve un objeto ArrayList cuyos elementos son copias del valor especificado.
			int	Cantidad de elementos	
Reverse	void	N / A	N / A	N / A	Invierte el orden de los elementos en el objeto ArrayList.
SetRange	void	N / A	int	Índice desde donde comenzar	Copia los elementos de una colección en un intervalo de elementos en el objeto ArrayList.
			ICollection	Colección a copiar	
Sort	void	N / A	IComparer	La implementación de IComparer a usar o nulo para usar la de cada elemento	Ordena los elementos del objeto ArrayList.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Synchronized	IList	Un objeto del tipo IList seguro para threads (thread safe)	IList	Un objeto del tipo IList para asegurar respecto de los threads (thread safe)	Devuelve un contenedor de ArrayList que está sincronizado.
ToArray	Object[]	Un vector de objetos contenidos en el ArrayList	N / A	N / A	Copia los elementos del objeto ArrayList en una nueva matriz.
TrimToSize	void	N / A	N / A	N / A	Establece la capacidad en el número de elementos en el objeto ArrayList.

Ejemplo

C#

```
namespace listas
{
    class Program
    {
        static void Main(string[] args)
        {
            IList lista = new ArrayList();
            lista.Add("uno");
            lista.Add("segundo");
            lista.Add("3ro");
            lista.Add(4);
            lista.Add(5.0F);
            lista.Add("segundo"); // duplicado, se agrega
            lista.Add(4); // duplicado, se agrega
            foreach(Object o in lista)
            Console.WriteLine(o);

            Console.WriteLine("----------------------");
            Console.WriteLine("Usando el indexador");
            Console.WriteLine("----------------------");

            for(int i = 0; i < lista.Count; i++)
                 Console.WriteLine(lista[i]);
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
```

```
    Sub Main()
        Dim lista As IList = New ArrayList()
        lista.Add("uno")
        lista.Add("segundo")
        lista.Add("3ro")
        lista.Add(4)
        lista.Add(5.0F)
        lista.Add("segundo") ' duplicado, se agrega
        lista.Add(4) ' duplicado, se agrega
        For Each o As Object In lista
            Console.WriteLine(o)
        Next

        Console.WriteLine("----------------------")
        Console.WriteLine("Usando el indexador")
        Console.WriteLine("----------------------")

        For i As Integer = 0 To lista.Count - 1
            Console.WriteLine(lista(i))
        Next
        Console.ReadKey()
    End Sub
End Module
```

Se deben tener en cuenta dos hechos importantes en el ejemplo: se agregan los duplicados y se realiza autoboxing con los elementos por valor.

Stack

La clase System.Collections.Stack es una clase de colección que proporciona acceso controlado a sus elementos. Una pila funciona según el principio de Último Entrado Primero Salido – UEPS (Last In First Out – LIFO), lo que significa que el último elemento insertado en la pila será el primer elemento a ser retirado de ella. Pilas y colas son estructuras de datos muy comunes en informática y su aplicación también lo es tanto en hardware como software. La inserción de un elemento en la pila que se denomina 'Push' (empujar), mientras que cuando se quita un elemento de la pila que se llama 'Pop' (extraer). Cuando al objeto no se lo elimina, sino que sólo se lee desde la parte superior de la pila, se lo denomina una operación 'Peek' (mirar).

Los constructores que admite la clase se muestran en la siguiente tabla:

Constructor	Descripción
Stack()	Inicializa una nueva instancia de la clase Stack que está vacía y tiene la capacidad inicial predeterminada.
Stack(ICollection)	Inicializa una nueva instancia de la clase Stack que contiene elementos copiados de la colección especificada y que tiene la misma capacidad inicial que el número de elementos copiados.

Constructor	Descripción
Stack(Int32)	Inicializa una nueva instancia de la clase Stack que está vacía y tiene la capacidad inicial especificada o la que es por defecto utilizando la mayor.

Stack implementa todos los métodos y propiedades de las interfaces ICollection, IEnumerable e ICloneable. La siguiente tabla describe algunos de los principales métodos de la clase:

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Clear	void	N / A	N / A	N / A	Quita todos los objetos del objeto del tipo Stack.
Contains	bool	Verdadero (true o True) si se encuentra el objeto.	Object	El objeto a localizar. Puede ser nulo	Determina si el objeto especificado está en el objeto del tipo Stack.
Peek	Object	El objeto a principio de la pila.	N / A	N / A	Retorna el objeto en la parte superior del objeto del tipo Stack, pero no lo elimina.
Pop	Object	El objeto removido.	N / A	N / A	Quita y devuelve el objeto en la parte superior del objeto del tipo Stack.
Push	void	N / A	Object	El objeto a colocar en la pila.	Inserta un objeto en la parte superior del objeto del tipo Stack.
Synchronized	Stack	La pila sincronizada	Stack	La pila a sincronizar	Devuelve una pila sincronizada asegurada de threads (threads safe).
ToArray	Object[]	Un nuevo vector de objetos con los elementos de la pila.	N / A	N / A	Copia los elementos del objeto del tipo Stack en un nuevo vector.

Ejemplo

C#

```
namespace pilas
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack stack = new Stack();
            stack.Push(2);
            stack.Push(4);
```

```
            stack.Push(6);
            while (stack.Count != 0)
            {
                Console.WriteLine(stack.Pop());
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim stack As New Stack()
        stack.Push(2)
        stack.Push(4)
        stack.Push(6)
        While stack.Count <> 0
            Console.WriteLine(stack.Pop())
        End While
        Console.ReadKey()
    End Sub
End Module
```

Queue

Una cola funciona según el principio Primero Entrado Primero Salido – PEPS (First In First Out – FIFO), lo que significa que el primer elemento insertado en la cola será el primer elemento retirado de ella. “Poner en cola” un elemento es insertarlo en la cola, y la eliminación de un elemento de la cola que se denomina “quitar de la cola”. Como en una pila, hay también una operación de lectura, donde el elemento no se elimina sino sólo se lee desde la parte frontal de la cola.

La clase Queue implementa una cola como un vector circular en el que se puede insertar objetos en un extremo y los quita por el otro. La capacidad de un objeto Queue es el número de elementos que dicho objeto puede contener. Al agregarle elementos a un objeto de cola, el objeto Queue automáticamente aumenta su capacidad según sea necesario mediante la reasignación. Para disminuir la capacidad de un objeto Queue, invocar el método TrimToSize.

Los constructores que admite la clase se muestran en la siguiente tabla:

Constructor	Descripción
Queue()	Inicializa una nueva instancia de la clase Queue que está vacía, tiene la capacidad inicial predeterminada y usa un factor de crecimiento por defecto.

Constructor	Descripción
Queue(ICollection)	Inicializa una nueva instancia de la clase Queue que contiene elementos copiados de la colección especificada, que tiene la misma capacidad inicial que el número de elementos copiados y usa un factor de crecimiento por defecto.
Queue(Int32)	Inicializa una nueva instancia de la clase Queue que está vacía, tiene la capacidad inicial especificada y usa un factor de crecimiento por defecto.
Queue(Int32, Single)	Inicializa una nueva instancia de la clase Queue que está vacía, tiene la capacidad inicial indicada en el primer argumento y usa un factor de crecimiento indicado en el segundo.

Queue implementa todos los métodos y propiedades de las interfaces ICollection, IEnumerable e ICloneable.

En la siguiente tabla se describen algunos de los métodos públicos más importantes de la clase Queue.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Clear	void	N / A	N / A	N / A	Quita todos los objetos de la cola.
Contains	bool	Verdadero (true o True) si se encuentra el objeto.	Object	El objeto a localizar. Puede ser nulo	Determina si el objeto especificado está en la cola.
Dequeue	Object	El objeto removido.	N / A	N / A	Quita y devuelve el objeto al comienzo del objeto de cola.
Enqueue	void	N / A	Object	El objeto a colocar en la cola.	Agrega un objeto al final del objeto de cola.
Peek	Object	El objeto a principio de la cola.	N / A	N / A	Devuelve el objeto al principio de la cola, pero no elimina el objeto.
Synchronized	Queue	La cola sincronizada	Queue	La cola a sincronizar	Devuelve una cola sincronizada asegurada de threads (threads safe).

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
ToArray	Object[]	Un nuevo vector de objetos con los elementos de la cola	N / A	N / A	Copia los elementos del objeto Queue en un nuevo vector.
TrimToSize	void	N / A	N / A	N / A	Establece la capacidad en el número de elementos en la cola.

Ejemplo

C#

```
namespace colas
{
    class Program
    {
        static void Main(string[] args)
        {
            Queue queue = new Queue();
            queue.Enqueue(2);
            queue.Enqueue(4);
            queue.Enqueue(6);
            while (queue.Count != 0)
            {
                Console.WriteLine(queue.Dequeue());
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim queue As New Queue()
        queue.Enqueue(2)
        queue.Enqueue(4)
        queue.Enqueue(6)
        While queue.Count <> 0
            Console.WriteLine(queue.Dequeue())
        End While
        Console.ReadKey()
    End Sub
End Module
```

Diccionarios

Los diccionarios son una especie de colección de objetos almacenados como pares clave-valor. Cada valor de la colección se identifica por su clave. Todas las claves de la colección son únicas y no puede haber más de una clave del mismo tipo. Los dos tipos más comunes de Diccionarios en el espacio de nombres System.Collections son Hashtable y SortedList.

Hashtable

Hashtable almacena objetos como pares clave-valor. Cada elemento (o valor) se encuentra almacenado como un objeto y se identifica por su clave. Una tabla hash almacena la clave y su valor como una estructura del tipo DictionaryEntry. La clave para utilizar correctamente esta colección es seleccionar adecuadamente el objeto que actúa como clave. Para que dicho objeto sea utilizado correctamente como clave, asegurarse de sobrescribir los métodos Equals()(o la interfaz IComparer) y GetHashCode() (o la interfaz IHashCodeProvider) que el objeto clave hereda de Object, de modo que:

- Equals() compruebe la igualdad de instancia en lugar de la igualdad de referencia predeterminada y seguir las relaciones de equivalencia descriptas para la comparación de objetos.
- GetHashCode () retorne un número entero igual para instancias que sean iguales de la clase.
- Los valores devueltos por GetHashCode() se distribuyen equitativamente entre el MinValue y MaxValue para el tipo Int32.

La clase Hashtable implementa las interfaces ICollection IDictionary, IEnumerable, ISerializable, IDeserializationCallback e ICloneable. Por este motivo, implementa todas las propiedades y métodos que se definen en las interfaces mencionadas.

Muchos constructores de la clase se consideran actualmente obsoletos, por eso se enumeran los más importantes, que son:

Constructor	Descripción
Hashtable()	Inicializa una nueva instancia vacía de la clase Hashtable utilizando la capacidad inicial, el factor de carga, el proveedor de código hash y el comparador predeterminados.
Hashtable(IDictionary)	Inicializa una nueva instancia de la clase Hashtable copiando los elementos del diccionario especificado al nuevo objeto Hashtable. El nuevo objeto mencionado tiene una capacidad inicial igual al número de elementos copiados y utiliza el factor de carga, el proveedor de código hash y el comparador por defecto.

Constructor	Descripción
Hashtable(IEqualityComparer)	Inicializa una nueva instancia vacía de la clase Hashtable utilizando la capacidad inicial y el factor de carga predeterminada además del objeto IEqualityComparer especificado.
Hashtable(Int32)	Inicializa una nueva instancia vacía de la clase Hashtable utilizando la capacidad inicial especificada y el factor de carga, el proveedor de código hash y el comparador por defecto.
Hashtable(IDictionary, IEqualityComparer)	Inicializa una nueva instancia de la clase Hashtable copiando los elementos del diccionario especificado en un objeto Hashtable nuevo. El nuevo objeto Hashtable tiene una capacidad inicial igual al número de elementos copiados, y utiliza el factor de carga por defecto y el objeto IEqualityComparer especificado.
Hashtable(IDictionary, Single)	Inicializa una nueva instancia de la clase Hashtable copiando los elementos del diccionario especificado al nuevo objeto Hashtable. El objeto nuevo tiene una capacidad inicial igual al número de elementos copiados y utiliza el factor de carga especificado, y el proveedor de código hash y comparador predeterminados.
Hashtable(Int32, IEqualityComparer)	Inicializa una nueva instancia vacía de la clase Hashtable utilizando la capacidad inicial y el objeto IEqualityComparer especificados junto con el factor de carga predeterminado.
Hashtable(Int32, Single)	Inicializa una nueva instancia vacía de la clase Hashtable utilizando la capacidad inicial y el factor de carga especificada, junto con el proveedor de código hash y el comparador predeterminados.
Hashtable(IDictionary, Single, IEqualityComparer)	Inicializa una nueva instancia de la clase Hashtable copiando los elementos del diccionario especificado al objeto Hashtable nuevo. El nuevo objeto tiene una capacidad inicial igual al número de elementos copiados y utiliza el factor de carga y el objeto IEqualityComparer especificados.
Hashtable(Int32, Single, IEqualityComparer)	Inicializa una nueva instancia vacía de la clase Hashtable utilizando la capacidad inicial, el factor de carga, y el objeto IEqualityComparer especificados.

En la siguiente tabla se describen algunos de los métodos públicos más importantes de la clase:

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
ContainsKey	bool	Verdadero si se encuentra el objeto	Object	La clave a encontrar	Determina si el objeto Hashtable contiene una clave específica.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
ContainsValue	bool	Verdadero si se encuentra el objeto	Object	El valor a encontrar	Determina si el objeto Hashtable contiene un valor específico.
Synchronized	Hashtable	La tabla sincronizada	Hashtable	La tabla a sincronizar	Devuelve un contenedor de Hashtable que se sincroniza.

Se debe tener en cuenta que el enumerador que se debe utilizar con un objeto del tipo Hashtable es un IDictionaryEnumerator.

Los métodos para manipular los objetos contenidos en la tabla hash más utilizados son los definidos en la interfaz IDictionary.

Tener cuidado al utilizar la colección de no volver a utilizar una clave que se haya usado previamente. Si este fuera el caso, se lanzaría una excepción del tipo ArgumentException. Por lo tanto, si existiese la posibilidad de repetir un valor clave, controlar dicha excepción.

Ejemplo

C#

```
namespace hash
{
    class Program
    {
        static void Main(string[] args)
        {
            Hashtable tabla = new Hashtable();
            try
            {
                tabla.Add("uno", "1ro");
                tabla.Add("segundo", 2);
                tabla.Add("tercero", "3º");
                // Intentando usar la misma clave
                tabla.Add("tercero", "III");
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("La clave está en uso: " + e.Message);
            }
            // Devuelve el conjunto de objetos que son las claves
            ICollection claves = tabla.Keys;
            Console.WriteLine("-- Imprimiendo claves --");
            foreach (Object c in claves)
            {
                Console.WriteLine("Clave: " + c.ToString());
            }
```

```
            // Devuelve el conjunto de objetos que son los valores
            ICollection coleccion = tabla.Values;
            Console.WriteLine("-- Imprimiendo valores --");
            foreach (Object c in coleccion)
            {
                Console.WriteLine("Valor: " + c.ToString());
            }

            Console.WriteLine("-- Imprimiendo pares clave / valor --");
            foreach (DictionaryEntry de in tabla)
            {
                Console.WriteLine("Clave: " + de.Key + "\t Valor: " + de.Value);
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim tabla As New Hashtable()
        Try
            tabla.Add("uno", "1ro")
            tabla.Add("segundo", 2)
            tabla.Add("tercero", "3º")
            ' Intentando usar la misma clave
            tabla.Add("tercero", "III")
        Catch e As ArgumentException
            Console.WriteLine("La clave está en uso: " + e.Message)
        End Try
        ' Devuelve el conjunto de objetos que son las claves
        Dim claves As ICollection = tabla.Keys
        Console.WriteLine("-- Imprimiendo claves --")
        For Each c As Object In claves
            Console.WriteLine("Clave: " + c.ToString())
        Next

        ' Devuelve el conjunto de objetos que son los valores
        Dim coleccion As ICollection = tabla.Values
        Console.WriteLine("-- Imprimiendo valores --")
        For Each c As Object In coleccion
            Console.WriteLine("Valor: " + c.ToString())
        Next

        Console.WriteLine("-- Imprimiendo pares clave / valor --")
        For Each de As DictionaryEntry In tabla
            Console.WriteLine(
              "Clave: " + de.Key + vbTab + " Valor: " + de.Value.ToString)
        Next
        Console.ReadKey()
    End Sub
```

```
End Module
```

El programa genera la siguiente salida:

```
La clave está en uso: Item has already been added. Key in dictionary: 'tercero'
 Key being added: 'tercero'
-- Imprimiendo claves --
Clave: tercero
Clave: uno
Clave: segundo
-- Imprimiendo valores --
Valor: 3º
Valor: 1ro
Valor: 2
-- Imprimiendo pares clave / valor --
Clave: tercero   Valor: 3º
Clave: uno       Valor: 1ro
Clave: segundo   Valor: 2
```


SortedList

Esta clase maneja una lista ordenada y es similar a la tabla hash, la diferencia es que los elementos se ordenan de acuerdo con la clave. Una de las ventajas de utilizar un SortedList es que se puede obtener los elementos de la colección mediante un índice entero, al igual que se puede hacer con un vector. En el caso de SortedList, si se desea utilizar una clase propia como clave, entonces, además de las consideraciones descritas en Hashtable, también hay que asegurarse que la clase implementa la interfaz IComparable.

La clase String y otros tipos de datos primitivos proporcionan una implementación de esta interfaz y por lo tanto se pueden utilizar como claves en un SortedList directamente.

La clase SortedList implementa las interfaces IDictionary, ICollection, IEnumerable e ICloneable. Se puede acceder a un elemento de un objeto del tipo SortedList mediante su clave o por su índice, como se indicó anteriormente. Si se tiene acceso a un elemento por su clave, es similar a la forma en que se utiliza una implementación en un objeto del tipo IDictionary. Si se tiene acceso a un elemento por su índice, es similar a la forma en que se utiliza una implementación en un objeto del tipo IList.

La secuencia del índice se basa en la secuencia en la cual se ordena la lista. Cuando se agrega un elemento, se inserta en el objeto SortedList en el orden correcto, y la indexación se ajusta en consecuencia. Cuando se quita un elemento, la indexación también se ajusta en consecuencia. Por lo tanto, el índice de un determinado par clave / valor puede cambiar a medida que los elementos se agregan o quitan del objeto del tipo SortedList.

SortedList implementa todas las propiedades y métodos definidos en las interfaces que implementa. Además tiene también una propiedad Capacity que indica la capacidad actual del objeto SortedList.

Los constructores que admite la clase se muestran en la siguiente tabla:

Constructor	Descripción
SortedList()	Inicializa una nueva instancia de la clase SortedList que está vacía, tiene la capacidad inicial predeterminada y está ordenada de acuerdo con la interfaz IComparable implementada por cada clave agregada al objeto del tipo SortedList que se está creando.
SortedList(IComparer)	Inicializa una nueva instancia de la clase SortedList que está vacía, tiene la capacidad inicial predeterminada y está ordenada de acuerdo con el objeto que implementa la interfaz IComparer especificado.
SortedList(IDictionary)	Inicializa una nueva instancia de la clase SortedList que contiene elementos copiados del diccionario especificado, tiene la misma capacidad inicial que el número de elementos copiados y está ordenada de acuerdo con la interfaz IComparable implementada por cada clave.
SortedList(Int32)	Inicializa una nueva instancia de la clase SortedList que está vacía, tiene la capacidad inicial especificada y está ordenada de acuerdo con la interfaz IComparable implementada por cada clave agregada al objeto del tipo SortedList que se está creando.
SortedList(IComparer, Int32)	Inicializa una nueva instancia de la clase SortedList que está vacía, tiene la capacidad inicial especificada y está ordenada de acuerdo con el objeto que implementa la interfaz IComparer especificado.
SortedList(IDictionary, IComparer)	Inicializa una nueva instancia de la clase SortedList que contiene elementos copiados del diccionario especificado, tiene la misma capacidad inicial que el número de elementos copiados y está ordenada de acuerdo con el objeto que implementa la interfaz IComparer especificado.

En la tabla siguiente se describen algunos de los métodos públicos más importantes de la clase.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
ContainsKey	bool	Verdadero si se encuentra la clave	Object	El objeto a encontrar como clave	Determina si un objeto del tipo SortedList contiene una clave específica.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
ContainsValue	bool	Verdadero si se encuentra el valor. Puede ser nulo.	Object	El objeto a encontrar como clave. Puede ser nulo.	Determina si un objeto del tipo SortedList contiene un valor específico.
GetByIndex	Object	El objeto que es el valor almacenado	int	El índice del valor a obtener	Obtiene el valor en el índice especificado en un objeto del tipo SortedList.
GetKey	Object	El objeto que es la clave almacenada	int	El índice de la clave a obtener	Obtiene la clave en el índice especificado en un objeto del tipo SortedList.
GetKeyList	IList	Un objeto del tipo IList con las claves	N / A	N / A	Obtiene las claves de un objeto del tipo SortedList.
GetValueList	IList	Un objeto del tipo IList con los valores	N / A	N / A	Obtiene los valores de un objeto del tipo SortedList.
IndexOfKey	int	El valor del índice de la clave o -1 si no se encuentra	Object	La clave a encontrar	Devuelve el índice de base cero de la clave especificada en un objeto del tipo SortedList.
IndexOfValue	int	El valor del índice del objeto valor o -1 si no se encuentra	Object	El objeto valor a encontrar. Puede ser nulo	Devuelve el índice de base cero del valor especificado en un objeto del tipo SortedList.
RemoveAt	N / A	N / A	int	Índice del elemento a remover	Quita el elemento en el índice especificado de un objeto del tipo SortedList.
SetByIndex	N / A	N / A	int	índice del valor a guardar	Reemplaza el valor de un índice específico en un objeto del tipo SortedList.
			Object	El objeto que es el valor a guardar	
TrimToSize	N / A	N / A	N / A	N / A	Establece la capacidad en el número de elementos en el objeto del tipo SortedList.

El siguiente ejemplo muestro el uso de una lista ordenada y como trabajan los métodos comúnmente utilizados.

Ejemplo

C#

```
using estudiantes;

namespace listaOrdenada
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crea e inicializa un nuevo SortedList.
            SortedList listaOrdenada = new SortedList();

            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
            Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

            Horario h1 = new Horario(2, 1, 0, 2, 20);
            Horario h2 = new Horario(2, 2, 20, 3, 40);
            Horario h3 = new Horario(2, 3, 40, 5, 0);
            Horario h4 = new Horario(3, 2, 20, 3, 40);
            Horario h5 = new Horario(4, 3, 40, 5, 0);

            listaOrdenada.Add(h1, a1);
            listaOrdenada.Add(h2, a2);
            listaOrdenada.Add(h3, a3);
            listaOrdenada.Add(h4, a4);
            listaOrdenada.Add(h5, a5);
            try
            {
                Console.WriteLine("Intentando agregar una clave que existe: " + h5);
                listaOrdenada.Add(h5, a6);
            }
            catch(Exception e)
            {
                Console.WriteLine(e.Message);
            }

            // Mostrar propiedades y valores del SortedList.
            Console.WriteLine("Lista ordenada");
            Console.WriteLine("  Count:    {0}", listaOrdenada.Count);
            Console.WriteLine("  Capacity: {0}", listaOrdenada.Capacity);
            Console.WriteLine("  Claves y Valores:");
            ImprimirClavesYValores(listaOrdenada);

            Console.ReadKey();
        }

        public static void ImprimirClavesYValores(SortedList lista)
        {
```

```
            Console.WriteLine("\t-Clave-\t\t\t-Valor-");
            for (int i = 0; i < lista.Count; i++)
            {
                Console.WriteLine("\t{0}:\t{1}",
                    lista.GetKey(i), lista.GetByIndex(i));
            }
            Console.WriteLine();
        }
    }
}
```

VB

```
Imports estudiantes

Module Module1

    Sub Main()
        ' Crea e inicializa un nuevo SortedList.
        Dim listaOrdenada As New SortedList()

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        Dim h1 As New Horario(2, 1, 0, 2, 20)
        Dim h2 As New Horario(2, 2, 20, 3, 40)
        Dim h3 As New Horario(2, 3, 40, 5, 0)
        Dim h4 As New Horario(3, 2, 20, 3, 40)
        Dim h5 As New Horario(4, 3, 40, 5, 0)

        listaOrdenada.Add(h1, a1)
        listaOrdenada.Add(h2, a2)
        listaOrdenada.Add(h3, a3)
        listaOrdenada.Add(h4, a4)
        listaOrdenada.Add(h5, a5)
        Try
            Console.WriteLine(
              "Intentando agregar una clave que existe: " + h5.ToString)
            listaOrdenada.Add(h5, a6)
        Catch e As Exception
            Console.WriteLine(e.Message)
        End Try


        ' Mostrar propiedades y valores del SortedList.
        Console.WriteLine("Lista ordenada")
        Console.WriteLine("  Count:    {0}", listaOrdenada.Count)
        Console.WriteLine("  Capacity: {0}", listaOrdenada.Capacity)
        Console.WriteLine("  Claves y Valores:")
        ImprimirClavesYValores(listaOrdenada)

        Console.ReadKey()
```

```
    End Sub

    Public Sub ImprimirClavesYValores(lista As SortedList)
        Console.WriteLine(vbTab + "-Clave-" + vbTab + vbTab + vbTab + "-Valor-")
        For i As Integer = 0 To lista.Count - 1
            Console.WriteLine(vbTab + "{0}:" + vbTab + _
              "{1}", lista.GetKey(i), lista.GetByIndex(i))
        Next
        Console.WriteLine()
    End Sub
End Module
```

BitArray

La clase BitArray contiene una matriz de indicadores de bits. Cada bit se suele utilizar como bandera que representa un valor booleano.

Se utiliza si se tienen varios valores booleanos que se desean almacenar eficientemente o para crear máscaras de bits que se correlacionan con patrones de bits de almacenamiento (por ejemplo, los permisos y accesos en un sistema de archivos de un sistema operativo están conformados como patrones de bits que describen las funcionalidades que éstos admiten).

La clase BitArray implementa las interfaces ICollection, IEnumerable e ICloneable. Cuando se crea un objeto del tipo BitArray, se especifica el número de bits que se deben almacenar. A continuación, se puede acceder a un bit en el objeto BitArray por su posición como si fuese un vector de base cero.

BitArray implementa todas las propiedades y métodos definidos en las interfaces. En la siguiente tabla se muestran los constructores de la clase:

Constructor	Descripción
BitArray(BitArray)	Inicializa una nueva instancia de la clase BitArray que contiene los valores de bit copiados de la colección del tipo BitArray especificada como argumento.
BitArray(Boolean [])	Inicializa una nueva instancia de la clase BitArray que contiene los valores de bit copiados del vector de booleanos.
BitArray(Byte [])	Inicializa una nueva instancia de la clase BitArray que contiene los valores de bit copiados del vector de bytes especificado como argumento.

Constructor	Descripción
BitArray(Int32)	Inicializa una nueva instancia de la clase BitArray que puede contener el número especificado de valores de bit, establecidos inicialmente en false.
BitArray(Int32 [])	Inicializa una nueva instancia de la clase BitArray que contiene los valores de bit copiados del vector de enteros de 32 bits.
BitArray(Int32, Boolean)	Inicializa una nueva instancia de la clase BitArray que puede contener el número especificado de valores de bit tomados del primer argumento, establecidos inicialmente en el valor especificado en el segundo.

En la siguiente tabla se describen algunas de las propiedades adicionales públicas de la clase BitArray:

Propiedad	Descripción
IsReadOnly	Devuelve un valor booleano que indica si el objeto del tipo BitArray es de sólo lectura.
Item	Obtiene o establece el valor del bit en la posición especificada en el objeto del tipo BitArray.
Length	Obtiene o establece el número de elementos en el objeto del tipo BitArray.

En la siguiente tabla se describen algunos de los métodos públicos de la clase:

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
And	BitArray	La instancia actual con el resultado de la operación realizada	BitArray	Objeto del tipo BitArray con el que realizar la operación	Realiza una operación AND bit a bit en los elementos del objeto del tipo BitArray actual respecto de los elementos correspondientes a otro objeto del tipo BitArray.
Get	bool	Valor del bit en la posición que indicó el argumento	int	Índice del bit a obtener	Obtiene el valor del bit en la posición especificada en el objeto del tipo BitArray.

Método	Retorna	Significa	Parámetros	Argumento Recibido	Descripción
Not	BitArray	La instancia actual con el resultado de la operación realizada	N / A	N / A	Invierte todos los valores de los bits en el objeto del tipo BitArray actual, de modo que los elementos que se encuentran en true se cambian en false y los elementos que están en false se cambian en true.
Or	BitArray	La instancia actual con el resultado de la operación realizada	BitArray	Objeto del tipo BitArray con el que realizar la operación	Realiza una operación OR bit a bit en los elementos del objeto BitArray actual respecto de los elementos correspondientes a otro objeto BitArray.
Set	N / A	N / A	int	Índice del bit a cambiar	Establece los bits en la posición especificada en el objeto del tipo de BitArray en el valor indicado.
			bool	Valor booleano del bit	
SetAll	N / A	N / A	bool	Valor booleano para asignar a todos los bits	Establece todos los bits en el objeto del tipo BitArray en el valor especificado.
Xor	BitArray	La instancia actual con el resultado de la operación realizada	BitArray	Objeto del tipo BitArray con el que realizar la operación	Realiza una operación OR exclusiva bit a bit en los elementos del objeto del tipo BitArray actual respecto de los elementos correspondientes a otro objeto del tipo BitArray.

Ejemplo

C#

```
namespace bits
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear una BitArray vacío con una longitud especifica.
            // A continuación, establecer el valor de los bits.
            BitArray ba1 = new BitArray(3);
            ba1[0] = true;
            ba1[1] = false;
            ba1[2] = true;
            // Crear un BitArray e inicializar los bits de inmediato.
            BitArray ba2 = new BitArray(new bool[] { true, true, false });
            // Realizar operaciones bit a bit.
            ba1.And(ba2); // Genera ba1: true, false, false.
```

```
            // Iterar sobre los bits de un BitArray.
            Console.WriteLine("AND");
            foreach (bool bit in ba1)
            {
                Console.Write("{0} ", bit);
            }
            ba1.Or(ba2); // Genera ba1: true, true, false.
            // Iterar sobre los bits de un BitArray.
            Console.WriteLine();
            Console.WriteLine("OR");
            foreach (bool bit in ba1)
            {
                Console.Write("{0} ", bit);
            }
            ba1.Xor(ba2); // Genera ba1: false, false, false.
            // Iterar sobre los bits de un BitArray.
            Console.WriteLine();
            Console.WriteLine("XOR");
            foreach (bool bit in ba1)
            {
                Console.Write("{0} ", bit);
            }
            // Especificar bits.
            ba1[2] = true; // Genera ba1: false, false, true.
            ba2.SetAll(true); // Genera ba2: true, true, true.
            Console.WriteLine();
            Console.WriteLine("Cambiar el tercer bit");
            // Iterar sobre los bits de un BitArray.
            foreach (bool bit in ba1)
            {
                Console.Write("{0} ", bit);
            }

            Console.Read();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        ' Crear una BitArray vacío con una longitud especifica.
        ' A continuación, establecer el valor de los bits.
        Dim ba1 As New BitArray(3)
        ba1(0) = True
        ba1(1) = False
        ba1(2) = True
        ' Crear un BitArray e inicializar los bits de inmediato.
        Dim ba2 As New BitArray(New Boolean() {True, True, False})
        ' Realizar operaciones bit a bit.
        ba1.And(ba2) ' Genera ba1: True, False, False.
        Console.WriteLine("AND")
        ' Iterar sobre los bits de un BitArray.
        For Each bit As Boolean In ba1
```

```
            Console.Write("{0} ", bit)
        Next
        ba1.Or(ba2) ' Genera ba1: True, True, False.
        Console.WriteLine()
        Console.WriteLine("OR")
        ' Iterar sobre los bits de un BitArray.
        For Each bit As Boolean In ba1
            Console.Write("{0} ", bit)
        Next
        ba1.Xor(ba2) ' Genera ba1: False, False, False.
        Console.WriteLine()
        Console.WriteLine("XOR")
        ' Iterar sobre los bits de un BitArray.
        For Each bit As Boolean In ba1
            Console.Write("{0} ", bit)
        Next
        ' Especificar bits.
        ba1(2) = True ' Genera ba1: False, False, True.
        ba2.SetAll(True) ' Genera ba2: True, True, True.
        Console.WriteLine()
        Console.WriteLine("Cambiar el tercer bit")
        ' Iterar sobre los bits de un BitArray.
        For Each bit As Boolean In ba1
            Console.Write("{0} ", bit)
        Next
        Console.Read()
    End Sub
End Module
```

Colecciones genéricas

La versión 2.0 del Framework de .NET introduce colecciones genéricas. Las colecciones genéricas se definen en términos de parámetros de tipo y son fuertemente tipadas. Cuando se crea una instancia de una clase de colección genérica, se debe especificar el parámetro para el tipo de objeto que la colección gestioná en su interior actuando como una clase contenedora. A partir de de ese punto cuando se opera con la colección, como por ejemplo, al agregar, quitar o acceder objetos contenidos dentro de ella, se utiliza el tipo de datos específicado en la creación de la instancia. Esto está en contraste con las clases no genéricas, las cuales se basan en objetos y almacenan referencias a elementos del tipo System.Object.

Las clases de colección genéricas se definen en el espacio de nombres System.Collections.Generic. La mayoría de de éstas proporcionan funcionalidad similar a sus homólogas no genéricas en el espacio de nombres System.Collections. Por lo general, se utilizan las clases de colección genéricas en lugar de las no genéricas. Las clases genéricas ofrecen los beneficios de la seguridad de tipos y un mejor rendimiento, ya que no requieren del estilo autoboxing o unboxing que consumen recursos.

Interfaces para colecciones genéricas

Nota: En los diagramas UML de las interfaces genéricas sólo se apreciará es uso del parámetro

genérico por medio de su uso en propiedades y métodos. Cuando la interfaz sea implementada en una clase concreta o el parámetro de tipo se utilice en una estructura, el diagrama lo muestra como un cuadrado superpuesto al diagrama de clase.

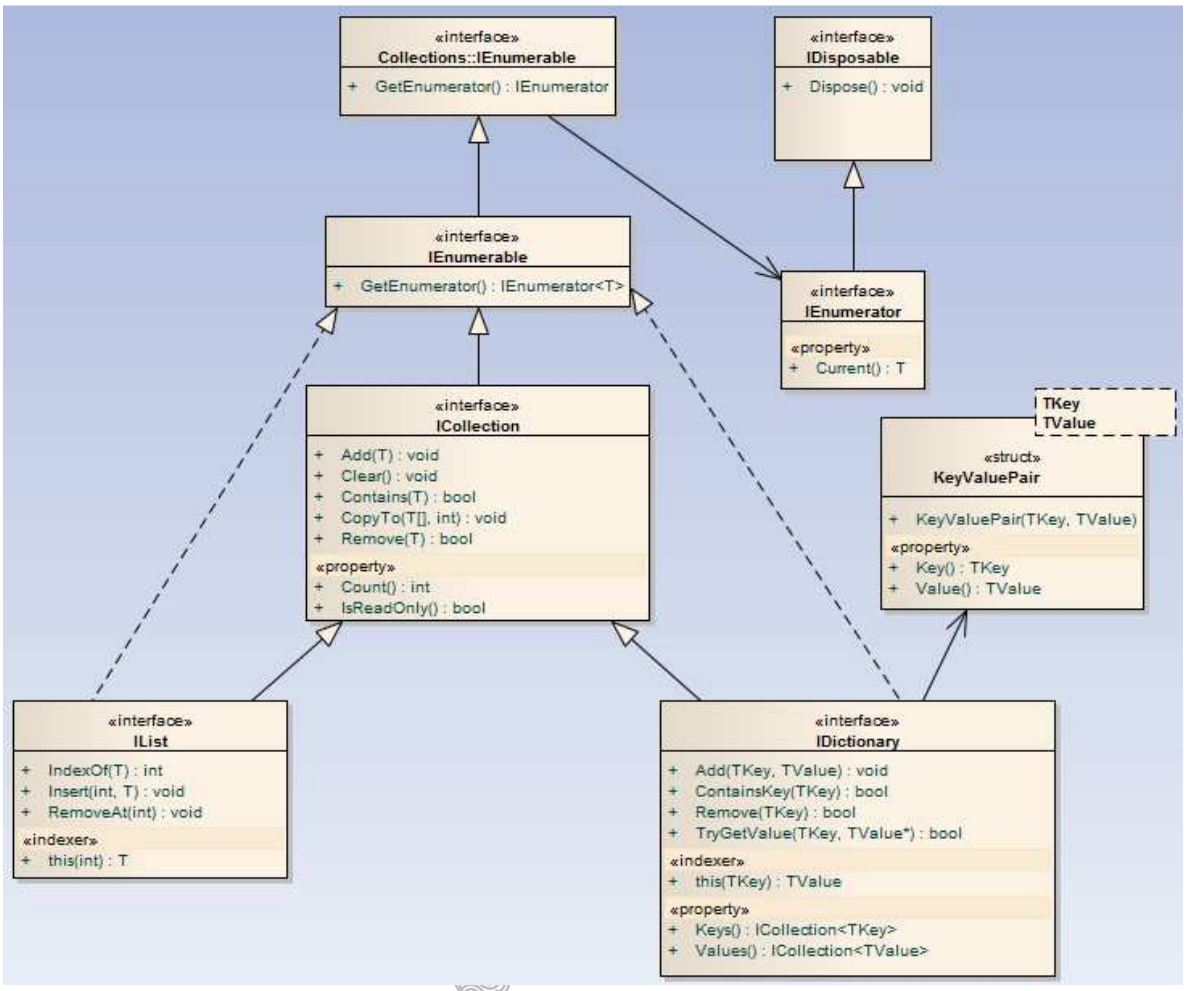
El espacio de nombres System.Collections.Generic define las interfaces genéricas que especifican el comportamiento de la mayoría de las colecciones genéricas. Se pueden organizar las interfaces genéricas, al igual que como se hizo anteriormente, en tres grupos, a saber:

- Interfaces de colección genéricas
- Interfaces de comparación genéricas
- Interfaces de enumeración genéricas

Interfaces de colección genéricas

Nota: se puede observar en las definiciones de los parámetros de tipo de algunas interfaces las palabras claves `in` y `out`, las cuales están asociadas a los conceptos de covarianza y contravariaza que se explicarán posteriormente

El siguiente diagrama UML es un resumen de las interfaces que se pueden agrupar bajo esta categoría:



Nota: las flechas punteadas indicas "realización" y es el lugar donde se implementan los métodos por rescritura de la interfaz. En otras palabras, es donde se realiza la implementación de la interfaz.

El espacio de nombres System.Collections.Generic define cuatro interfaces de colección genéricas, a saber:

- ```
  public interface IEnumerable<out T> : IEnumerable
  Public Interface IEnumerable(Of Out T)
                  Inherits IEnumerable
  ```
  Define un método GetEnumerator, que retorna un objeto que implementa la interfaz IEnumerator genérica. Se puede utilizar un enumerador para recorrer iterando sobre los elementos de cualquier colección genérica. El parámetro genérico esta definido como covariante, lo cual implica que se pueden definir subtipos del parámetro que se proporcione cuando se implementa la interfaz.
- ```
  public interface ICollection<T> : IEnumerable<T>, IEnumerable
  Public Interface ICollection(Of T)
  ```

```
        Inherits IEnumerable(Of T), IEnumerable
```

Hereda de las interfaces genérica y no genérica IEnumerable, y define las propiedades y métodos que se pueden utilizar con todas las colecciones genéricas.

- `public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable`
  `Public Interface IList(Of T)`
- `        Inherits ICollection(Of T), IEnumerable(Of T), IEnumerable`

Hereda de la interfaz ICollection genérica y representa una colección de objetos que se puede acceder individualmente por índice. Además de las propiedades que se definen en la interfaz ICollection genérica, IList define una propiedad Item que obtiene o establece el elemento en una posición con un índice en base a cero.

```
public interface IDictionary<TKey, TValue> : ICollection<KeyValuePair<TKey,
TValue>>, IEnumerable<KeyValuePair<TKey, TValue>>, IEnumerable
Public Interface IDictionary(Of TKey, TValue)
        Inherits ICollection(Of KeyValuePair(Of TKey, TValue)),  _
        IEnumerable(Of KeyValuePair(Of TKey, TValue)), IEnumerable
```

Hereda de la interfaz ICollection genérica y representa una colección de objetos pares clave / valor. Cada entrada en un objeto IDictionary es una estructura KeyValuePair genérica que representa una clave y un valor de un tipo especificado como parámetro. Para tener acceso a los objetos que se utilizan como clave o valor, se deben usar las propiedades Key y Value de la estructura. El conjunto de claves del diccionario se puede obtener como un objeto del tipo ICollection con la propiedad Keys. Con cada clave que pertenece a la colección obtenida se puede recorrer el diccionario utilizando el indexador.

Interfaces de comparación genéricas

El espacio de nombres System.Collections.Generic define dos interfaces genéricas de comparación a saber:

- `public interface IComparer<in T>`
  `Public Interface IComparer(Of In T)`

  Define un método de comparación que compara dos objetos de un tipo especificado y devuelve un valor que indica si uno es menor que, igual a, o mayor que el otro. La aplicación define lo que significa para los objetos a ser menor que, igual a, o mayor que la otra. El parámetro de tipo está definido como contravariante, lo cual implica que se pueden definir supertipos del parámetro que se proporcione cuando se implementa la interfaz.

  La clase de colección genérica List tiene los métodos Sort y BinarySearch que usa como parámetro una instancia que implementa la interfaz genérica IComparer para especificar la forma de ordenar la colección y realizar las búsquedas de los elementos. La implementación predeterminada de la interfaz genérica IComparer es la clase genérica Comparer. Para la comparación de cadenas, es conveniente usar colecciones más específicas diseñadas para tal fin como la clase StringComparer en lugar de Comparer<String> o Comparer(Of String) (C# o VB). StringComparer define las propiedades y métodos que permiten realizar comparaciones de cadenas mediante el uso de diferentes combinaciones de sensibilidad respecto de la cultura (la ubicación geográfica en la cual se realiza la ejecución del programa) tanto como diferenciamiento entre mayúsculas y minúsculas.

- `public interface IEqualityComparer<in T>`
  `Public Interface IEqualityComparer(Of In T)`
  Define un método Equals que determina si los dos objetos de un tipo especificado son iguales. La interfaz también define un método GetHashCode que devuelve un código hash para dicho objeto.
  La clase de colección genérica Dictionary tiene constructores que toman una instancia de la interfaz IEqualityComparer genérica como parámetro para especificar cómo el diccionario compara las claves para determinar la igualdad. La implementación predeterminada de la interfaz genérica IEqualityComparer es la clase genérica EqualityComparer. Para las comparaciones de igualdad de cadenas, es conveniente usar colecciones más específicas diseñadas para tal fin como la clase la clase StringComparer en lugar de EqualityComparer<String> o EqualityComparer(Of String) (C# o VB).

El espacio de nombres System define una interfaz genérica que permite a un objeto compararse a sí mismo contra otro objeto del mismo tipo:

- `public interface IComparable<in T>`
  `Public Interface IComparable (Of In T)`
  Define un método CompareTo que compara el objeto actual contra otro objeto y devuelve un valor para indicar si el actual es menor que, igual a o mayor que el otro objeto. El parámetro genérico esta definido como covariante, lo cual implica que se pueden definir subtipos del parámetro que se proporcione cuando se implementa la interfaz. La interfaz genérica IComparable es implementada por tipos cuyos valores se pueden ordenar, como numéricos o cadenas. Se puede implementar la interfaz genérica IComparable en clases y estructuras propias para crear un método de comparación específico del tipo que sea adecuado para fines personalizados de manera que los métodos de ordenamiento de las distintas clases que utilicen la interfaz realicen comparaciones en base a la especificada.

Interfaces de enumeración genéricas

El espacio de nombres System.Collections.Generic define la siguientes interfaces de enumeración:

- `public interface IEnumerator<out T> : IDisposable, IEnumerator`
  `Public Interface IEnumerator(Of Out T)`
  `Inherits IDisposable, IEnumerator`
  Soporta la enumeración en una colección genérica. Se puede obtener una instancia de la interfaz genérica IEnumerator al invocar al método de una colección genérica GetEnumerator cuando dicho objeto sea del tipo ICollection. El parámetro genérico esta definido como covariante, lo cual implica que se pueden definir subtipos del parámetro que se proporcione cuando se implementa la interfaz.
  La interfaz genérica IEnumerator define una propiedad Current que indica el elemento actual de la colección. La interfaz hereda los métodos MoveNext y Reset de la interfaz no genérica IEnumerator.
- `public interface IEnumerable<out T> : IEnumerable`
  `Public Interface IEnumerable(Of Out T)`
  `Inherits IEnumerable`
  Muchas interfaces y las clases del espacio de nombres System.Collections.Generic heredan la interfaz IEnumerable<T> para implementar enumeradores. Si bien se explicó está interfaz anteriormente para una mejor explicación de las restantes interfaces explicadas, el lugar correcto en el cuál se la debe clasificar es en esta sección.

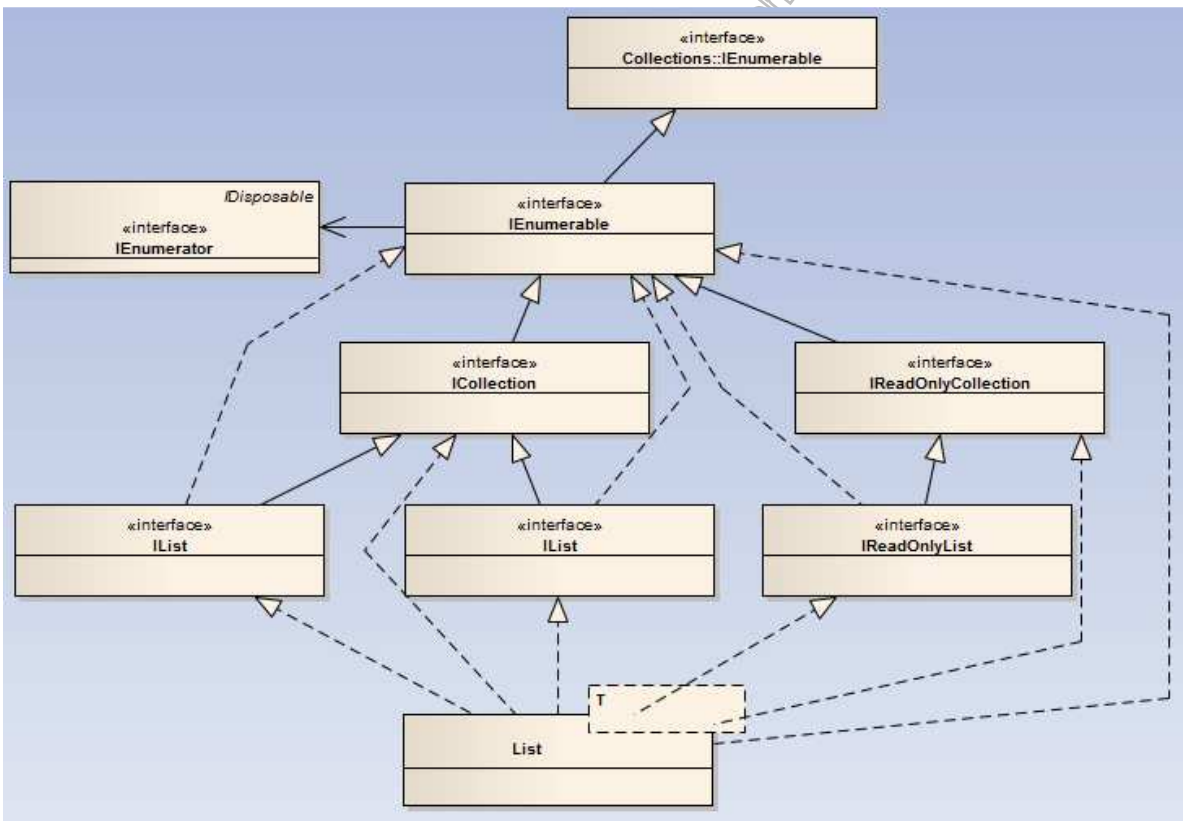
Clases genéricas para colecciones

List

La clase genérica List<T> o List(Of T) (C# o VB) representa una lista de objetos con tipos seguros (también conocidos como *fuertes* por estar especificado el tipo en tiempo de compilación) que se puede acceder mediante un índice, y proporciona métodos para buscar, ordenar y manipular listas. List<T> o List(Of T) (C# o VB) implementa la interfaz genérica IList<T> o IList(Of T) (C# o VB) utilizando un vector cuyo tamaño aumenta dinámicamente según se lo requiera.

La clase de lista es el equivalente genérico de la clase ArrayList no genérica. Cuando se almacenan los tipos por referencia, el comportamiento de las dos clases es idéntico. Al almacenar tipos por valor, la lista ofrece un mejor rendimiento, ya que evita las operaciones de boxing y unboxing al agregar, quitar o acceder a los elementos.

El siguiente diagrama UML muestra la clase con las respectivas interfaces que implementa:



El siguiente diagrama muestra la implementación de la clase:

La clase List<T> o List(Of T) (C# o VB) utiliza un comparador para la igualdad y un comparador para el orden, de la siguiente manera:

- Los métodos como Contains, IndexOf, LastIndexOf y Remove utilizan un comparador de igualdad para los elementos de la lista. Si el tipo del elemento (objeto) que se encuentra en la lista implementa la interfaz genérica IEquatable, el comparador de igualdad es el método IEquatable.Equals, de lo contrario se utiliza el comparador de igualdad predeterminado, el cual es el método Object.Equals.

- Los métodos como BinarySearch y Sort utilizan un comparador para el orden de los elementos de la lista. Si el tipo de elemento (objeto) que se encuentra en la lista implementa la interfaz genérica o no genérica IComparable, el comparador usado es el método genérico o no genérico IComparable.CompareTo, de lo contrario, se debe proporcionar un comparador y un delegado de comparación para que la clase de lista pueda usar para ordenar.

La clase genérica List<T> o List(Of T) (C# o VB) rescribe todas las propiedades y métodos definidos en las interfaces que implementa. También define una propiedad Capacity que obtiene o establece el número total de elementos que la estructura de datos interna (el vector que puede cambiar de tamaño) esta en condiciones de almacenar sin que se cambie el tamaño.

Ejemplo

C#

```
using estudiantesT;

namespace listasT
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Alumno> alumnos = new List<Alumno>();

            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);


            Console.WriteLine("\nCapacity: {0}", alumnos.Capacity);

            alumnos.Add(a1);
            alumnos.Add(a2);
            alumnos.Add(a3);
            alumnos.Add(a4);
            alumnos.Add(a5);

            Console.WriteLine();
            foreach (Alumno alumno in alumnos)
            {
                Console.WriteLine(alumno);
            }

            Console.WriteLine("\nCapacity: {0}", alumnos.Capacity);
            Console.WriteLine("Count: {0}", alumnos.Count);

            Console.WriteLine("\nContains(\"{1}\"): {0}",
                alumnos.Contains(a1), a1);
```

```
            Console.WriteLine(
        "Insertando un objeto igual al primero de la lista en el tercer lugar");
            Alumno a6 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Console.WriteLine("\nInsert(2, \"{0}\")", a6);
            alumnos.Insert(2, a6);

            Console.WriteLine();
            foreach (Alumno alumno in alumnos)
            {
                Console.WriteLine(alumno);
            }

            Console.WriteLine("\nalumnos[3]: {0}", alumnos[3]);

            Console.WriteLine("\nRemove(\"{0}\")", a6);
            alumnos.Remove(a6);

            Console.WriteLine();
            foreach (Alumno alumno in alumnos)
            {
                Console.WriteLine(alumno);
            }

            alumnos.TrimExcess();
            Console.WriteLine("\nTrimExcess()");
            Console.WriteLine("Capacity: {0}", alumnos.Capacity);
            Console.WriteLine("Count: {0}", alumnos.Count);

            alumnos.Sort();
            Console.WriteLine();
            Console.WriteLine("Escribiendo la lista ordenada");

            foreach (Alumno alumno in alumnos)
            {
                Console.WriteLine(alumno);
            }

            alumnos.Clear();
            Console.WriteLine("\nClear()");
            Console.WriteLine("Capacity: {0}", alumnos.Capacity);
            Console.WriteLine("Count: {0}", alumnos.Count);

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

VB

```
Imports estudiantesT

Module Module1

    Sub Main()
        Dim alumnos As New List(Of Alumno)
```

```
Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)

Console.WriteLine(vbLf & "Capacity: {0}", alumnos.Capacity)

alumnos.Add(a1)
alumnos.Add(a2)
alumnos.Add(a3)
alumnos.Add(a4)
alumnos.Add(a5)

Console.WriteLine()
For Each alumno As Alumno In alumnos
    Console.WriteLine(alumno)
Next

Console.WriteLine(vbLf & "Capacity: {0}", alumnos.Capacity)
Console.WriteLine("Count: {0}", alumnos.Count)

Console.WriteLine(vbLf & "Contains(""{1}""): {0}", _
    alumnos.Contains(a1), a1)

Console.WriteLine(
     "Insertando un objeto igual al primero de la lista en el tercer lugar")
Dim a6 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
Console.WriteLine(vbLf & "Insert(2, ""{0}"")", a6)
alumnos.Insert(2, a6)

Console.WriteLine()
For Each alumno As Alumno In alumnos
    Console.WriteLine(alumno)
Next

Console.WriteLine(vbLf & "alumnos(3): {0}", alumnos(3))

Console.WriteLine(vbLf & "Remove(""{0}"")", a6)
alumnos.Remove(a6)

Console.WriteLine()
For Each alumno As Alumno In alumnos
    Console.WriteLine(alumno)
Next

alumnos.TrimExcess()
Console.WriteLine(vbLf & "TrimExcess()")
Console.WriteLine("Capacity: {0}", alumnos.Capacity)
Console.WriteLine("Count: {0}", alumnos.Count)

alumnos.Sort()
Console.WriteLine()
Console.WriteLine("Escribiendo la lista ordenada")
```

```
        For Each alumno As Alumno In alumnos
            Console.WriteLine(alumno)
        Next

        alumnos.Clear()
        Console.WriteLine(vbLf & "Clear()")
        Console.WriteLine("Capacity: {0}", alumnos.Capacity)
        Console.WriteLine("Count: {0}", alumnos.Count)

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

LinkedList

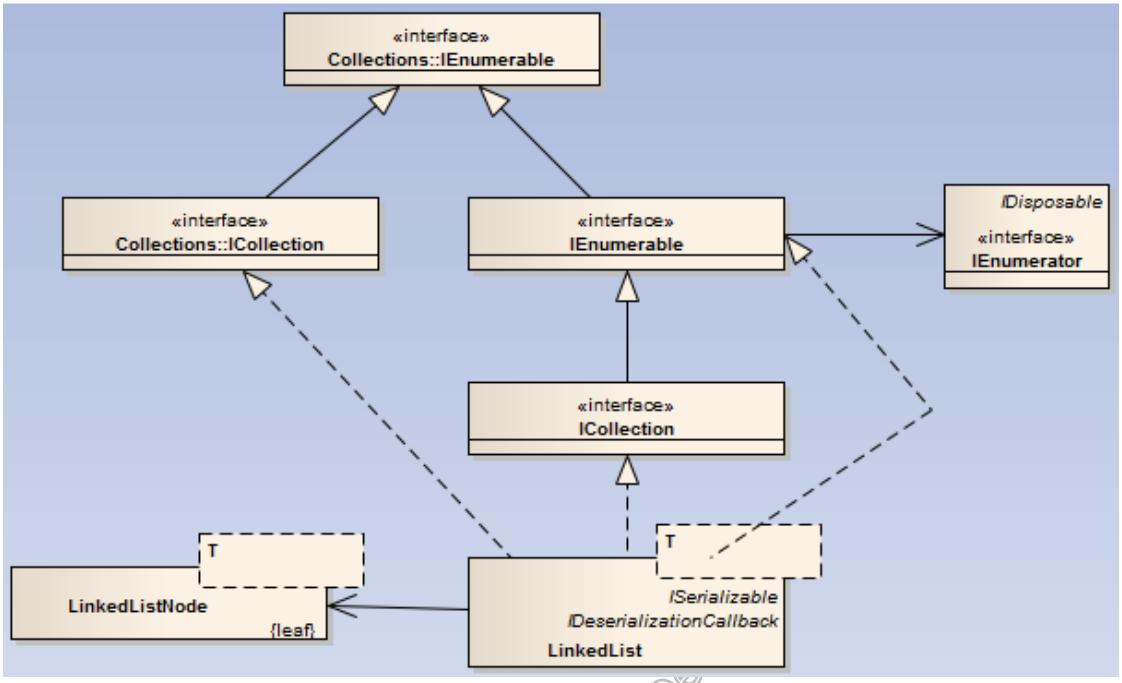
La clase genérica LinkedList<T> o LinkedList (Of T) (C# o VB) representa una lista doblemente enlazada de objetos de tipos asegurados, que se puede recorrer en cualquier dirección en orden secuencial. A diferencia de la clase genérica List<T> o List(Of T) (C# o VB), la clase genérica LinkedList<T> o LinkedList (Of T) (C# o VB) no proporciona métodos para buscar u ordenar los elementos o para acceder a elementos por índice con un idexador.

Normalmente, se utiliza una clase genérica LinkedList<T> o LinkedList (Of T) (C# o VB) cuando se debe almacenar objetos fuertemente tipados en orden, pero en la que no se desea tener acceso a los elementos por su posición en un determinado índice. Es compatible con los enumeradores e implementa la interfaz ICollection, lo que le otorga consistencia con otras clases de colección en el Framework de .NET.

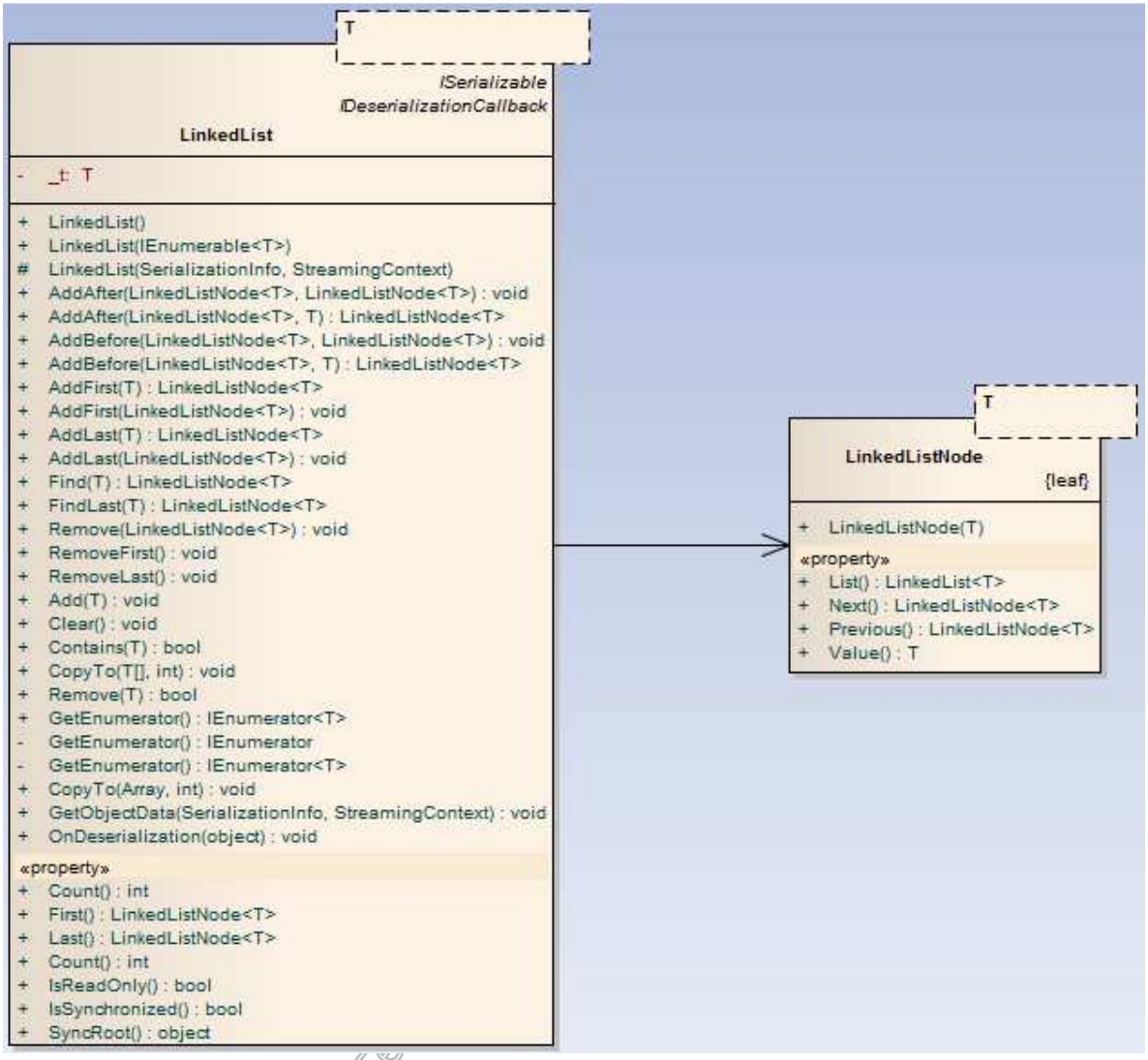
Se pueden eliminar nodos y volver a colocarlos, ya sea en la misma lista o en otra lista, lo que resulta en no crear ningún objeto adicional en el heap si se conserva la referencia asociada al objeto en cuestión que conforma el nodo. Debido a que la lista también mantiene un contador interno, la propiedad Count es una operación rápida ya que no se recorre la lista para conocer el contenido. Cada nodo de un objeto LinkedList<T> o LinkedList (Of T) (C# o VB) es del tipo LinkedListNode<T> o LinkedListNode (Of T) (C# o VB). Debido a que LinkedList<T> o LinkedList (Of T) (C# o VB) está doblemente enlazada, cada nodo apunta hacia el siguiente nodo y hacia atrás al nodo anterior.

Las listas que contienen los tipos por referencia se desempeñan mejor cuando un nodo y su valor se crean al mismo tiempo. LinkedList<T> o LinkedList (Of T) (C# o VB) acepta valores nulos como un valor válido para la propiedad Value en los tipos por referencia y admite valores duplicados. Si LinkedList<T> o LinkedList (Of T) (C# o VB) está vacía, las propiedades First y Last contienen un valor nulo.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama muestra la implementación de la clase así como también el uso de la clase LinkedListNode<T> o LinkedListNode (Of T) (C# o VB). Cada elemento de la colección LinkedList<T> o LinkedList (Of T) (C# o VB) es un LinkedListNode<T> o LinkedListNode (Of T) (C# o VB). El LinkedListNode<T> o LinkedListNode (Of T) (C# o VB) contiene un valor, una referencia a la lista enlazada a la que pertenece, almacenada en una referencia al siguiente nodo, y una al nodo anterior.



Cada nodo en una clase genérica LinkedList es un objeto LinkedListNode genérico. Cada objeto en una lista LinkedListNode tiene propiedades Next y Previous que indican los nodos siguiente y anterior en la lista respecto de la posición actual en la que se encuentre.

El primer nodo de la lista tiene una propiedad Previous nula, y el último nodo de la lista tiene una propiedad Next nula. Cada nodo tiene una propiedad Value con el tipo asegurado que obtiene el valor en el nodo y una propiedad List que obtiene el objeto del tipo LinkedList a la que pertenece el nodo.

Ejemplo

C#

```
using estudiantesT;
```

```
namespace listaEnlazadaT
{
    class Program
    {
        static void Main(string[] args)
        {
            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);

            // Crear una nueva lista enlazada.
            Alumno[] alumnos = { a1, a2, a3, a4, a3, a1 };
            LinkedList<Alumno> clase = new LinkedList<Alumno>(alumnos);
            Imprimir(clase, "Valores de la lista enlazada:");
            ImprimeNodo("a3", a3);
            Console.WriteLine("clase.Contains(\"a3\") = {0}",
                clase.Contains(a3));

            // Agregar a5 al comienzo de la lista.
            clase.AddFirst(a5);
            ImprimeNodo("a5", a5);
            Imprimir(clase,
              "Verificación 1: Agregar 'a5' al principio de la lista:");

            // Mover el primer nodo para que sea el último.
            LinkedListNode<Alumno> nodo = clase.First;
            clase.RemoveFirst();
            clase.AddLast(nodo);
            Imprimir(clase,
              "Verificación 2: Mover el primer nodo para que sea el último:");

            Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);
            // Cambiar el último nodo para que sea a6.
            clase.RemoveLast();
            clase.AddLast(a6);
            ImprimeNodo("a6", a6);
            Imprimir(clase, "Verificación 3: Cambiar el último nodo por 'a6':");

            // Mover el último nodo para que sea el primero.
            nodo = clase.Last;
            clase.RemoveLast();
            clase.AddFirst(nodo);
            Imprimir(clase,
              "Verificación 4: Mover el último nodo para que sea el primero:");

            // Usando los paréntesis buscar la última ocurrencia de a3.
            clase.RemoveFirst();
            LinkedListNode<Alumno> actual = clase.FindLast(a3);
            ImprimeNodo("a3", a3);
            ApuntarANodo(actual,
              "Verificación 5: Buscar la última ocurrencia de 'a3':");

            // Agregar a7 y a8 después de a3 (el nodo actual)
```

```
Alumno a7 = new Alumno("Alejandro", "César", "Corso", 21, 21212);
Alumno a8 = new Alumno("Pablo", "Ésteban", "Pegas", 27, 27272);
clase.AddAfter(actual, a7);
clase.AddAfter(actual, a8);
ImprimeNodo("a3", a3);
ImprimeNodo("a7", a7);
ImprimeNodo("a8", a8);
ApuntarANodo(actual,
  "Verificación 6: Se agregó a7 y a8 después de a3:");

// Posicionarse en el nodo a4.
actual = clase.Find(a4);
ApuntarANodo(actual, "Verificación 7: Indicar el nodo a4:");
ImprimeNodo("a4", a4);

// Agregar a9 y a10 antes de a4
Alumno a9 = new Alumno("Inés", "Amelia", "García", 29, 29292);
Alumno a10 = new Alumno("Sandra", "Mónica", "Piedras", 23, 23232);
clase.AddBefore(actual, a9);
clase.AddBefore(actual, a10);
ImprimeNodo("a9", a9);
ImprimeNodo("a10", a10);
ImprimeNodo("a4", a4);
ApuntarANodo(actual,
  "Verificación 8: Agregar 'a9' y 'a10' antes de 'a4':");

// Mantener una referencia al nodo actual a4
// y al nodo anterior en la lista a10. Buscar el nodo a2
nodo = actual;
LinkedListNode<Alumno> nodo2 = actual.Previous;
actual = clase.Find(a2);
ImprimeNodo("a2", a2);
ApuntarANodo(actual, "Verificación 9: Apuntar al nodo a2:");

// El método AddBefore lanza una InvalidOperationException
// si se trata de agregar un nodo que ya está en la lista
Console.WriteLine("Verificación 10: Lanzar una excepción al " +
  "agregar el nodo (a4) que está en la lista:");
ImprimeNodo("a4", a4);
try
{
    clase.AddBefore(actual, nodo);
}
catch (InvalidOperationException ex)
{
    Console.WriteLine("Mensaje de la excepción: {0}", ex.Message);
}
Console.WriteLine();

// Sacar el nodo al que apunta 'nodo' y luego
// agregarlo antes del nodo actual.
// Indicar el nodo como el actual
clase.Remove(nodo);
clase.AddBefore(actual, nodo);
ImprimeNodo("actual", actual.Value);
```

```
ImprimeNodo("referenciado", nodo.Value);
ApuntarANodo(actual, "Verificación 11: Mover el nodo referenciado" +
  " (a4) antes del nodo actual (a2):");

// Remover el nodo actual.
clase.Remove(actual);
ImprimeNodo("actual", actual.Value);
ApuntarANodo(actual, "Verificación 12: Remover en nodo actual " +
  " (a2) y tratar de indicarlo:");

// Agregar un nodo después del referenciado por nodo
clase.AddAfter(nodo, actual);
ImprimeNodo("actual", actual.Value); // a2
ImprimeNodo("nodo", nodo.Value);  // a4
ApuntarANodo(actual, "Verificación 13: Agregar el nodo borrado " +
  "en la verificación 11 luego del referenciado:");

// El método Remove busca y borra el primer
// nodo que tenga el valor especificado
clase.Remove(a3);
ImprimeNodo("a3", a3);
Imprimir(clase, "Verificación 14: Remover el primer nodo " +
  "a3 encontrado:");

// Cuando el tipo de una lista enlazada se convierte
// a un tipo ICollection<Alumno>, el método Add agrega
// el nodo al final de la lista.
clase.RemoveLast();
ICollection<Alumno> icoll = clase;
Alumno a11 = new Alumno("Claudia", "Silvina", "Peralta", 30, 30303);
icoll.Add(a11);
ImprimeNodo("a11", a11);
Imprimir(clase, "Verificación 15: Quitar el último nodo, " +
  "convertir a ICollection, y agregar 'a11':");

Console.WriteLine("Verificación 16: Copiar la lista a un vector:");
// Crear un vector con el mismo número
// de elementos que hay en la lista
Alumno[] vecAlumnos = new Alumno[clase.Count];
clase.CopyTo(vecAlumnos, 0);

foreach (Alumno a in vecAlumnos)
{
    Console.WriteLine(a);
}

// Liberar todos los nodos.
clase.Clear();

Console.WriteLine();
Console.WriteLine("Verificación 17: Limpiar la lista enlazada. " +
  "Ver si está 'a1' = {0}",
     clase.Contains(a1));

Console.ReadLine();
```

```
        }

        private static void Imprimir(LinkedList<Alumno> alumnos, string mensaje)
        {
            Console.WriteLine(mensaje);
            foreach (Alumno alumno in alumnos)
            {
                Console.Write(alumno + " ");
            }
            Console.WriteLine();
            Console.WriteLine();
        }

        private static void ApuntarANodo(LinkedListNode<Alumno> nodo,
             string mensaje)
        {
            Console.WriteLine(mensaje);
            if (nodo.List == null)
            {
                Console.WriteLine("El nodo '{0}' no esta en la lista.\n",
                    nodo.Value);
                return;
            }

            StringBuilder resultado = new StringBuilder("(" + nodo.Value + ")");
            LinkedListNode<Alumno> nodeP = nodo.Previous;

            while (nodeP != null)
            {
                resultado.Insert(0, nodeP.Value + " ");
                nodeP = nodeP.Previous;
            }

            nodo = nodo.Next;
            while (nodo != null)
            {
                resultado.Append(" " + nodo.Value);
                nodo = nodo.Next;
            }

            Console.WriteLine(resultado);
            Console.WriteLine();
        }

        public static void ImprimeNodo(string s, Alumno a)
        {
            Console.WriteLine("Nodo {0}: {1}", s, a);
        }
    }
}
```

VB

```
Imports estudiantesT
Imports System.Text
```

```
Module Module1

    Sub Main()
        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)

        ' Crear una nueva lista enlazada.
        Dim alumnos() As Alumno = {a1, a2, a3, a4, a3, a1}
        Dim clase As New LinkedList(Of Alumno)(alumnos)
        Imprimir(clase, "Valores de la lista enlazada:")
        ImprimeNodo("a3", a3)
        Console.WriteLine("clase.Contains('a3') = {0}", clase.Contains(a3))

        ' Agregar a5 al comienzo de la lista.
        clase.AddFirst(a5)
        ImprimeNodo("a5", a5)
        Imprimir(clase, "Verificación 1: Agregar 'a5' al principio de la lista:")

        ' Mover el primer nodo para que sea el último.
        Dim nodo As LinkedListNode(Of Alumno) = clase.First
        clase.RemoveFirst()
        clase.AddLast(nodo)
        Imprimir(clase, _
              "Verificación 2: Mover el primer nodo para que sea el último:")

        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)
        ' Cambiar el último nodo para que sea a6.
        clase.RemoveLast()
        clase.AddLast(a6)
        ImprimeNodo("a6", a6)
        Imprimir(clase, "Verificación 3: Cambiar el último nodo por 'a6':")

        ' Mover el último nodo para que sea el primero.
        nodo = clase.Last
        clase.RemoveLast()
        clase.AddFirst(nodo)
        Imprimir(clase, _
              "Verificación 4: Mover el último nodo para que sea el primero:")

        ' Usando los paréntesis buscar la última ocurrencia de a3.
        clase.RemoveFirst()
        Dim actual As LinkedListNode(Of Alumno) = clase.FindLast(a3)
        ImprimeNodo("a3", a3)
        ApuntarANodo(actual, "Verificación 5: Buscar la última ocurrencia de 'a3':")

        ' Agregar a7 y a8 después de a3 (el nodo actual)
        Dim a7 As New Alumno("Alejandro", "César", "Corso", 21, 21212)
        Dim a8 As New Alumno("Pablo", "Ésteban", "Pegas", 27, 27272)
        clase.AddAfter(actual, a7)
        clase.AddAfter(actual, a8)
        ImprimeNodo("a3", a3)
        ImprimeNodo("a7", a7)
```

```
ImprimeNodo("a8", a8)
ApuntarANodo(actual, "Verificación 6: Se agregó a7 y a8 después de a3:")

' Posicionarse en el nodo a4.
actual = clase.Find(a4)
ApuntarANodo(actual, "Verificación 7: Indicar el nodo a4:")
ImprimeNodo("a4", a4)

' Agregar a9 y a10 antes de a4
Dim a9 As New Alumno("Inés", "Amelia", "García", 29, 29292)
Dim a10 As New Alumno("Sandra", "Mónica", "Piedras", 23, 23232)
clase.AddBefore(actual, a9)
clase.AddBefore(actual, a10)
ImprimeNodo("a9", a9)
ImprimeNodo("a10", a10)
ImprimeNodo("a4", a4)
ApuntarANodo(actual, "Verificación 8: Agregar 'a9' y 'a10' antes de 'a4':")

' Mantener una referencia al nodo actual a4
' y al nodo anterior en la lista a10. Buscar el nodo a2
nodo = actual
Dim nodo2 As LinkedListNode(Of Alumno) = actual.Previous
actual = clase.Find(a2)
ImprimeNodo("a2", a2)
ApuntarANodo(actual, "Verificación 9: Apuntar al nodo a2:")

' El método AddBefore lanza una InvalidOperationException
' si se trata de agregar un nodo que ya está en la lista
Console.WriteLine("Verificación 10: Lanzar una excepción " + _
     "al agregar el nodo (a4) que está en la lista:")
ImprimeNodo("a4", a4)
Try
    clase.AddBefore(actual, nodo)
Catch ex As InvalidOperationException
    Console.WriteLine("Mensaje de la excepción: {0}", ex.Message)
End Try

Console.WriteLine()

    ' Sacar el nodo al que apunta 'nodo' y luego
    ' agregarlo antes del nodo actual.
    ' Indicar el nodo como el actual
clase.Remove(nodo)
clase.AddBefore(actual, nodo)
ImprimeNodo("actual", actual.Value)
ImprimeNodo("referenciado", nodo.Value)
ApuntarANodo(actual, "Verificación 11: Mover el nodo referenciado " + _
     " (a4) antes del nodo actual (a2):")

    ' Remover el nodo actual.
clase.Remove(actual)
ImprimeNodo("actual", actual.Value)
ApuntarANodo(actual, "Verificación 12: Remover en nodo actual (a2) " + _
     "y tratar de indicarlo:")
```

```
        ' Agregar un nodo después del referenciado por nodo
        clase.AddAfter(nodo, actual)
        ImprimeNodo("actual", actual.Value) ' a2
        ImprimeNodo("nodo", nodo.Value) ' a4
        ApuntarANodo(actual, "Verificación 13: Agregar el nodo borrado en " + _
             "la verificación 11 luego del referenciado:")

        ' El método Remove busca y borra el primer
        ' nodo que tenga el valor especificado
        clase.Remove(a3)
        ImprimeNodo("a3", a3)
        Imprimir(clase, "Verificación 14: Remover el primer nodo a3 encontrado:")

        ' Cuando el tipo de una lista enlazada se convierte
        ' a un tipo ICollection<Alumno>, el método Add agrega
        ' el nodo al final de la lista.
        clase.RemoveLast()
        Dim icoll As ICollection(Of Alumno) = clase
        Dim a11 As Alumno = New Alumno("Claudia", "Silvina", "Peralta", 30, 30303)
        icoll.Add(a11)
        ImprimeNodo("a11", a11)
        Imprimir(clase, "Verificación 15: Quitar el último nodo, convertir " + _
             "a ICollection, y agregar 'a11':")

        Console.WriteLine("Verificación 16: Copiar la lista a un vector:")
        ' Crear un vector con el mismo número
        ' de elementos que hay en la lista
        Dim vecAlumnos(clase.Count) As Alumno
        clase.CopyTo(vecAlumnos, 0)

        For Each a As Alumno In vecAlumnos
            Console.WriteLine(a)
        Next

        ' Liberar todos los nodos.
        clase.Clear()

        Console.WriteLine()
        Console.WriteLine("Verificación 17: Limpiar la lista enlazada. Ver " + _
             "si está 'a1' = {0}", clase.Contains(a1))

        Console.ReadLine()
    End Sub

    Private Sub Imprimir(alumnos As LinkedList(Of Alumno), mensaje As String)
        Console.WriteLine(mensaje)
        For Each alumno As Alumno In alumnos
            Console.Write(alumno.ToString + " ")
        Next

        Console.WriteLine()
        Console.WriteLine()
    End Sub

    Private Sub ApuntarANodo(nodo As LinkedListNode(Of Alumno), mensaje As String)
```

```
        Console.WriteLine(mensaje)
        If nodo.List Is Nothing Then
            Console.WriteLine("El nodo '{0}' no esta en la lista." + vbNewLine,
                              nodo.Value)
            Return
        End If

        Dim resultado As New StringBuilder("(" + nodo.Value.ToString + ")")
        Dim nodeP As LinkedListNode(Of Alumno) = nodo.Previous

        While Not nodeP Is Nothing
            resultado.Insert(0, nodeP.Value.ToString + " ")
            nodeP = nodeP.Previous
        End While

        nodo = nodo.Next
        While Not nodo Is Nothing
            resultado.Append(" " + nodo.Value.ToString)
            nodo = nodo.Next
        End While
        Console.WriteLine(resultado)
        Console.WriteLine()
    End Sub

    Public Sub ImprimeNodo(s As String, a As Alumno)
        Console.WriteLine("Nodo {0}: {1}", s, a)
    End Sub
End Module
```

Queue

Como se explicó anteriormente, una cola funciona según el principio Primero Entrado Primero Salido – PEPS (First In First Out – FIFO), con la salvedad que en este caso al ser genérica sus tipos son seguros (tipos fuertes).

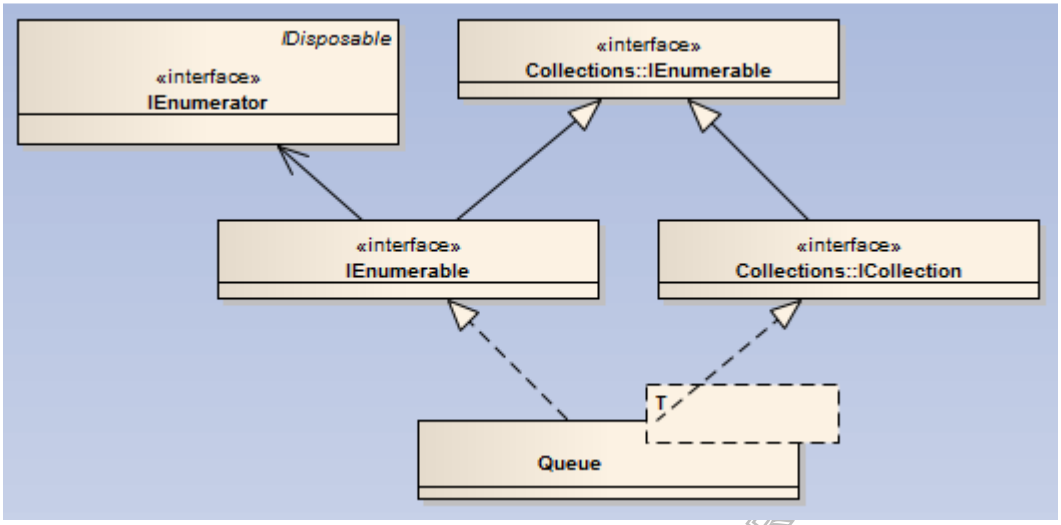
La clase genérica Queue<T> o Queue(Of T) (C# o VB) implementa una cola como un vector circular donde se inserta objetos en un extremo y se eliminan del otro.

Las colas son útiles para almacenar mensajes en el orden en que se recibieron para el tratamiento secuencial.

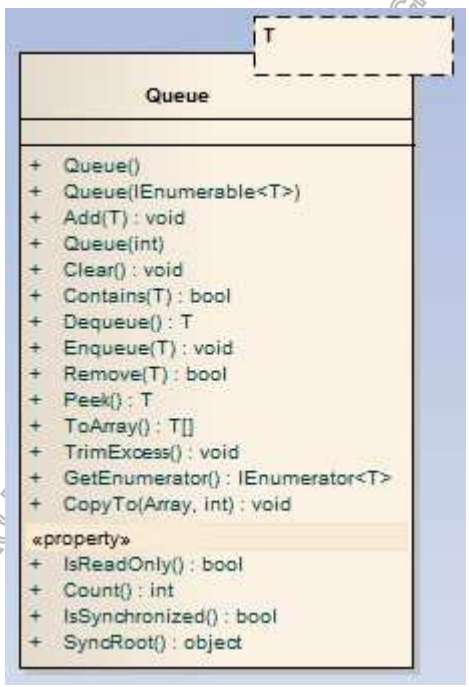
La capacidad de una Queue<T> o Queue(Of T) (C# o VB) es el número de elementos que puede contener. Cuando se le agregan elementos, la capacidad aumenta automáticamente según sea necesario mediante la reasignación de un vector interno. La capacidad se puede disminuir si se llama al método TrimExcess.

Queue<T> o Queue(Of T) (C# o VB) acepta nulos como valores válidos para los tipos por referencia que contiene y permite elementos duplicados.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra la implementación de la clase:



Ejemplo

C#

```
using estudiantesT;

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

```
{
    Queue<string> alumnos = new Queue<string>();

    Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
    Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
    Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
    Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
    Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
    Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

    alumnos.Enqueue(a1.ToString());
    alumnos.Enqueue(a2.ToString());
    alumnos.Enqueue(a3.ToString());
    alumnos.Enqueue(a4.ToString());
    alumnos.Enqueue(a5.ToString());
    alumnos.Enqueue(a6.ToString());

    Console.WriteLine("Contenido inicial de la cola:");

    // Una cola puede ser enumerada sin alterar su contenido
    foreach (string alumno in alumnos)
    {
        Console.WriteLine(alumno);
    }

    Console.WriteLine("\nSacando de la cola a '{0}'", alumnos.Dequeue());
    Console.WriteLine("Mirar el próximo elemento antes de sacarlo: {0}",
        alumnos.Peek());
    Console.WriteLine("Dequeuing '{0}'", alumnos.Dequeue());

    // Crear una copia de la cola usando el método ToArray junto al
    // constructor que acepta un objeto del tipo IEnumerable<T>
    Queue<string> copiaCola = new Queue<string>(alumnos.ToArray());

    Console.WriteLine("\nContenido de la primera copia:");
    foreach (string alumno in copiaCola)
    {
        Console.WriteLine(alumno);
    }

    // Crear un vector del doble del tamaño de la cola y copiar
    // los elementos que ésta almacena, comenzando por la mitad
    // del vector
    string[] vector2 = new string[alumnos.Count * 2];
    alumnos.CopyTo(vector2, alumnos.Count);

    // Crear una segunda cola, usando el método ToArray junto al
    // constructor que acepta un objeto del tipo IEnumerable<T>
    Queue<string> copiaCola2 = new Queue<string>(vector2);

    Console.WriteLine(
      "\nContenido de la segunda copia, con duplicados y valores nulos:");
    foreach (string alumno in copiaCola2)
    {
        Console.WriteLine(alumno);
```

```
            }

            Console.WriteLine("\ncopiaCola2.Contains(\"{0}\") = {1}",
                copiaCola2.Contains(a3.ToString()), a3);

            Console.WriteLine("\ncopiaCola2.Clear()");
            copiaCola2.Clear();
            Console.WriteLine("\ncopiaCola2.Count = {0}", copiaCola2.Count);
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System
Imports System.Collections.Generic
Imports estudiantesT
Module Module1

    Sub Main()

        Dim alumnos As New Queue(Of String)

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        alumnos.Enqueue(a1.ToString())
        alumnos.Enqueue(a2.ToString())
        alumnos.Enqueue(a3.ToString())
        alumnos.Enqueue(a4.ToString())
        alumnos.Enqueue(a5.ToString())
        alumnos.Enqueue(a6.ToString())

        Console.WriteLine("Contenido inicial de la cola:")

        ' Una cola puede ser enumerada sin alterar su contenido
        For Each _alumno As String In alumnos
            Console.WriteLine(_alumno)
        Next

        Console.WriteLine(vbNewLine + _
             "Sacando de la cola a '{0}'", alumnos.Dequeue())
        Console.WriteLine("Mirar el próximo elemento antes de sacarlo: {0}",
            alumnos.Peek())
        Console.WriteLine("Dequeuing '{0}'", alumnos.Dequeue())

        ' Crear una copia de la cola usando el método ToArray junto al
        ' constructor que acepta un objeto del tipo IEnumerable(Of T).
        Dim copiaCola As New Queue(Of String)(alumnos.ToArray())

        Console.WriteLine(vbNewLine + "Contenido de la primera copia:")
```

```
        For Each _alumno As String In copiaCola
            Console.WriteLine(_alumno)
        Next

        ' Crear un vector del doble del tamaño de la cola, compensando
        ' el hecho que Visual Basic asigna un elemento adicional
        ' al vector. Copiar los elementos de la cola, a partir de la
        ' mitad del vector.
        Dim vector2((alumnos.Count * 2) - 1) As String
        alumnos.CopyTo(vector2, alumnos.Count)

        ' Crear una segunda cola, usando el método ToArray junto al
        ' constructor que acepta un objeto del tipo IEnumerable(Of T).
        Dim copiaCola2 As New Queue(Of String)(vector2)

        Console.WriteLine(vbNewLine + _
            "Contenido de la segunda copia, con duplicados y valores nulos:")
        For Each _alumno As String In copiaCola2
            Console.WriteLine(_alumno)
        Next

        Console.WriteLine(vbLf & "copiaCola2.Contains(""{0}"") = {1}",
                copiaCola2.Contains(a3.ToString()), a3)

        Console.WriteLine(vbNewLine + "queueCopy.Clear()")
        copiaCola2.Clear()
        Console.WriteLine(vbNewLine + "queueCopy.Count = {0}", _
            copiaCola2.Count)
        Console.ReadKey()
    End Sub
End Module
```

Stack

La clase genérica Stack<T> o Stack(Of T) (C# o VB) implementa una pila como un vector circular, inserta objetos en un extremo y los elimina de la misma forma.

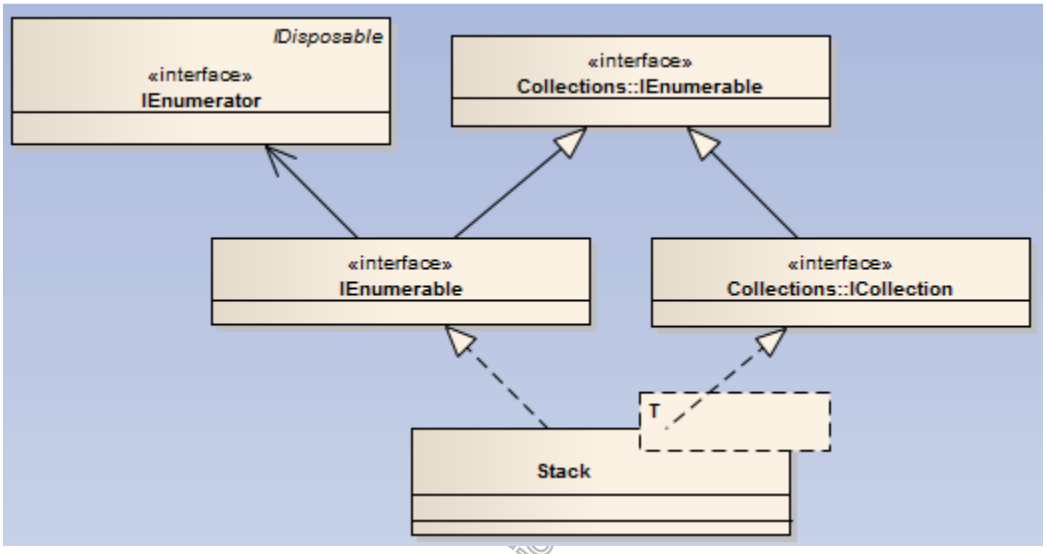
La capacidad de un Stack<T> o Stack(Of T) (C# o VB) es el número de elementos que éste puede contener. Cuando se agregan elementos a un Stack<T> o Stack(Of T) (C# o VB), la capacidad aumenta automáticamente según sea necesario mediante la reasignación del vector interno. La capacidad se puede disminuir si se llama TrimExcess. Este método se puede utilizar para minimizar la sobrecarga de la memoria que ocupa una colección de si no se añadirán elementos nuevos a la colección. El costo de reasignar y copiar una gran pila puede ser considerable. Sin embargo, el método TrimExcess no hace nada si la pila está ocupada en más de un 90% de la capacidad al momento de ejecutarlo. Esto evita incurrir en un costo grande de reasignación de memoria para una ganancia relativamente pequeña de recursos.

Si Count es menor que la capacidad de la pila, Push es una operación rápida. Si la capacidad necesita ser incrementada para el nuevo elemento, Push se convierte en una operación más lenta

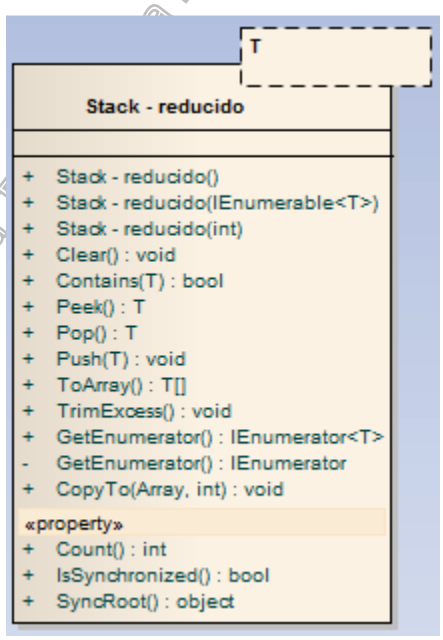
porque incluye la reasignación de los n elementos que posee en ese momento la pila. Pop es siempre una operación rápida.

Stack<T> o Stack(Of T) (C# o VB) acepta `null` o `Nothing` (C# o VB) como un valor válido para los tipos por referencia contenidos y permite elementos duplicados.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra una implementación de la clase:



Ejemplo

C#

```
using estudiantesT;

namespace pilaT
{
    class Program
    {
        static void Main(string[] args)
        {
            Stack<Alumno> alumnos = new Stack<Alumno>();

            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
            Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

            alumnos.Push(a1);
            alumnos.Push(a2);
            alumnos.Push(a3);
            alumnos.Push(a4);
            alumnos.Push(a5);
            alumnos.Push(a6);
            alumnos.Push(a1);
            alumnos.Push(a2);
            alumnos.Push(a3);
            alumnos.Push(a4);

            // Una pila se puede enumerar sin perturbar su contenido.
            foreach (Alumno alumno in alumnos)
            {
                Console.WriteLine(alumno);
            }

            Console.WriteLine("\nSacando de la pila '{0}'", alumnos.Pop());
            Console.WriteLine("Mirar el elemento que quedó para sacar: {0}",
                alumnos.Peek());
            Console.WriteLine("Sacando el elemento que se inspeccionó '{0}'",
              alumnos.Pop());

            // Crear una copia de la pila, utilizando el método ToArray y
            // el constructor que acepta un IEnumerable.
            Stack<Alumno> otraPila = new Stack<Alumno>(alumnos.ToArray());

            Console.WriteLine("\nContenido de la primera copia:");
            foreach (Alumno alumno in otraPila)
            {
                Console.WriteLine(alumno);
            }

            // Crear un vector de dos veces el tamaño de la pila y copiar
```

```
            // los elementos de la pila, a partir de la mitad del mismo
            Alumno[] vector = new Alumno[alumnos.Count * 2];
            alumnos.CopyTo(vector, alumnos.Count);

            // Se crea una segunda pila, utilizando el constructor que acepta un
            // IEnumerable<T> .
            Stack<Alumno> terceraPila = new Stack<Alumno>(vector);

            Console.WriteLine(
              "\nContenido de la segunda copia con duplicados y nulos:");
            foreach (Alumno alumno in terceraPila)
            {
                Console.WriteLine(alumno);
            }

            Console.WriteLine("\notraPila.Contains(\"a2\") = {0}",
                otraPila.Contains(a2));

            Console.WriteLine("\notraPila.Clear()");
            otraPila.Clear();
            Console.WriteLine("\notraPila.Count = {0}", otraPila.Count);
            Console.ReadKey();
        }
    }
}
```

VB

```
Imports System
Imports System.Collections.Generic
Imports estudiantesT
Module Module1

    Sub Main()

        Dim alumnos As New Queue(Of String)

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        alumnos.Enqueue(a1.ToString())
        alumnos.Enqueue(a2.ToString())
        alumnos.Enqueue(a3.ToString())
        alumnos.Enqueue(a4.ToString())
        alumnos.Enqueue(a5.ToString())
        alumnos.Enqueue(a6.ToString())

        Console.WriteLine("Contenido inicial de la cola:")

        ' Una cola puede ser enumerada sin alterar su contenido
        For Each _alumno As String In alumnos
            Console.WriteLine(_alumno)
```

```
        Next

        Console.WriteLine(vbNewLine + _
             "Sacando de la cola a '{0}'", alumnos.Dequeue())
        Console.WriteLine("Mirar el próximo elemento antes de sacarlo: {0}",
            alumnos.Peek())
        Console.WriteLine("Dequeuing '{0}'", alumnos.Dequeue())

        ' Crear una copia de la cola usando el método ToArray junto al
        ' constructor que acepta un objeto del tipo IEnumerable(Of T).
        Dim copiaCola As New Queue(Of String)(alumnos.ToArray())

        Console.WriteLine(vbNewLine + "Contenido de la primera copia:")
        For Each _alumno As String In copiaCola
            Console.WriteLine(_alumno)
        Next

        ' Crear un vector del doble del tamaño de la cola, compensando
        ' el hecho que Visual Basic asigna un elemento adicional
        ' al vector. Copiar los elementos de la cola, a partir de la
        ' mitad del vector.
        Dim vector2((alumnos.Count * 2) - 1) As String
        alumnos.CopyTo(vector2, alumnos.Count)

        ' Crear una segunda cola, usando el método ToArray junto al
        ' constructor que acepta un objeto del tipo IEnumerable(Of T).
        Dim copiaCola2 As New Queue(Of String)(vector2)

        Console.WriteLine(vbNewLine + _
            "Contenido de la segunda copia, con duplicados y valores nulos:")
        For Each _alumno As String In copiaCola2
            Console.WriteLine(_alumno)
        Next

        Console.WriteLine(vbLf & "copiaCola2.Contains(""{0}"") = {1}",
                copiaCola2.Contains(a3.ToString()), a3)

        Console.WriteLine(vbNewLine + "queueCopy.Clear()")
        copiaCola2.Clear()
        Console.WriteLine(vbNewLine + "queueCopy.Count = {0}", _
            copiaCola2.Count)
        Console.ReadKey()
    End Sub
End Module
```


HashSet

Un conjunto matemático es una colección de elementos que no contiene duplicados, y cuyos integrantes no están en ningún orden en particular. La clase HashSet<T> o HashSet(Of T) (C# o VB) se basa en el modelo matemático de conjuntos y proporciona operaciones de alto rendimiento sobre éstos similares a las de acceso a claves de colecciones que sean del tipo diccionario o del estilo de Hashtable.

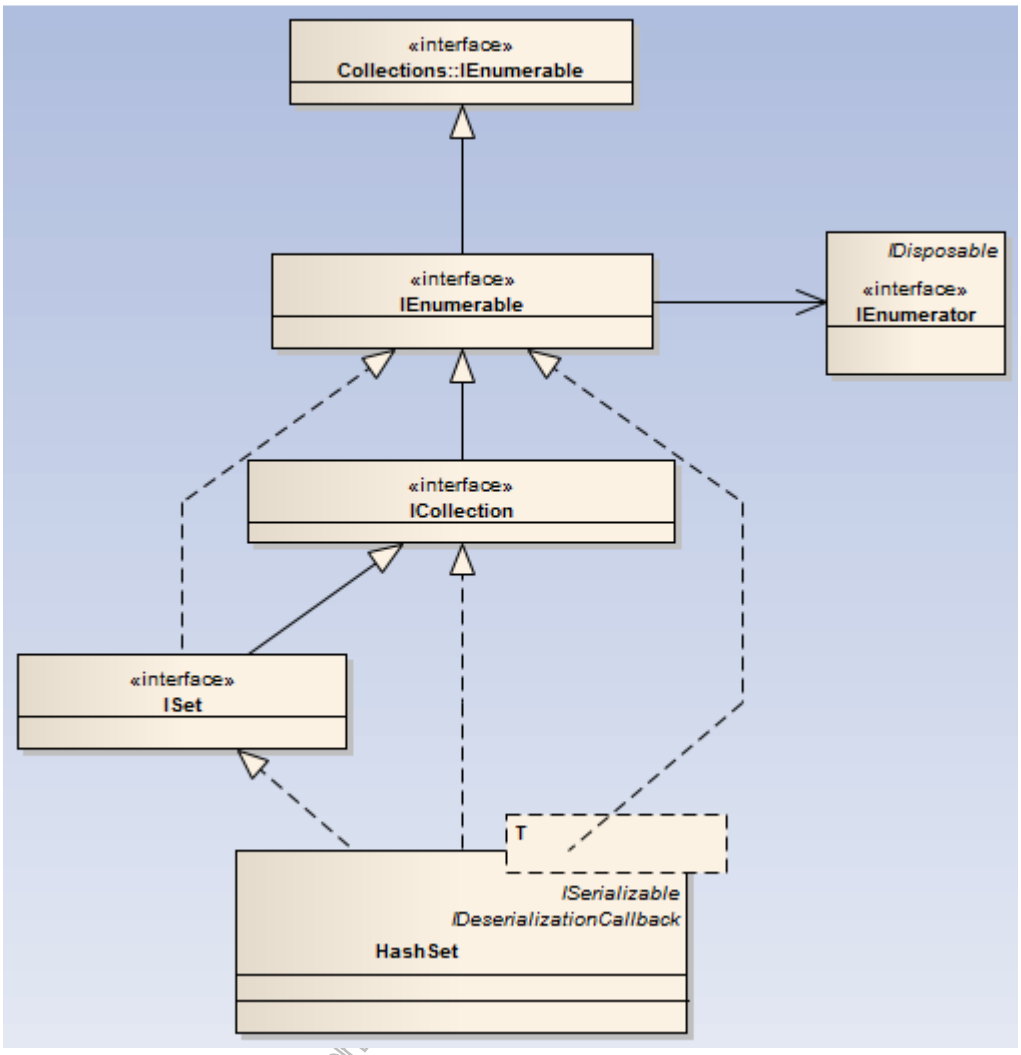
Una colección HashSet no está ordenada y no puede contener elementos duplicados. Si por cuestiones de diseño es necesario tener los elementos duplicados u ordenados y es más importante que el rendimiento de acceso o búsqueda, considerar el uso de la clase List junto con el método Sort que está posee. Inclusive, el método permite especificar el criterio de comparación a utilizar, lo cual le brinda aún mayor flexibilidad.

HashSet<T> o HashSet(Of T) (C# o VB) proporciona muchas operaciones similares a la de los conjuntos matemáticos, como agregar otros conjuntos (uniones) o quitarlos. Inclusive permite encontrar la intersección de dos conjuntos.

La capacidad de un objeto HashSet<T> o HashSet(Of T) (C# o VB) es el número de elementos que puede contener. La capacidad de un objeto de este tipo aumenta automáticamente a medida que se le agregan elementos.

A partir del marco. NET 4, HashSet<T> o HashSet(Of T) (C# o VB) implementa la interfaz ISet<T> o ISet(Of T) (C# o VB).

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra la implementación de la clase:

Ejemplo

C#

```
using estudiantesT;

namespace hashsetT
{
    class Program
    {
        static void Main(string[] args)
        {
            HashSet<Alumno> conjuntoOrdenado = new HashSet<Alumno>();

            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
```

```
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
            Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

            conjuntoOrdenado.Add(a1);
            conjuntoOrdenado.Add(a2);
            conjuntoOrdenado.Add(a3);
            conjuntoOrdenado.Add(a4);
            conjuntoOrdenado.Add(a5);
            conjuntoOrdenado.Add(a6);
            conjuntoOrdenado.Add(a2); // Duplicado. No se agrega
            conjuntoOrdenado.Add(a3); // Duplicado. No se agrega
            conjuntoOrdenado.Add(a1); // Duplicado. No se agrega
            conjuntoOrdenado.Add(a2); // Duplicado. No se agrega


            Console.WriteLine(
              "Ejemplo de conjunto que no admite duplicados y esta desordenado");
            MostrarLista(conjuntoOrdenado);

            Console.WriteLine("Cantidad de elementos en el conjunto: " +
              conjuntoOrdenado.Count);
            Console.ReadKey();
        }

        public static void MostrarLista(ISet<Alumno> conjunto)
        {
            foreach (Alumno elemento in conjunto)
            {
                Console.WriteLine(elemento);
            }
        }
    }
}
```

VB

```
Imports estudiantesT

Module Module1

    Sub Main()
        Dim conjuntoOrdenado As New HashSet(Of Alumno)()

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        conjuntoOrdenado.Add(a1)
        conjuntoOrdenado.Add(a2)
        conjuntoOrdenado.Add(a3)
        conjuntoOrdenado.Add(a4)
        conjuntoOrdenado.Add(a5)
```

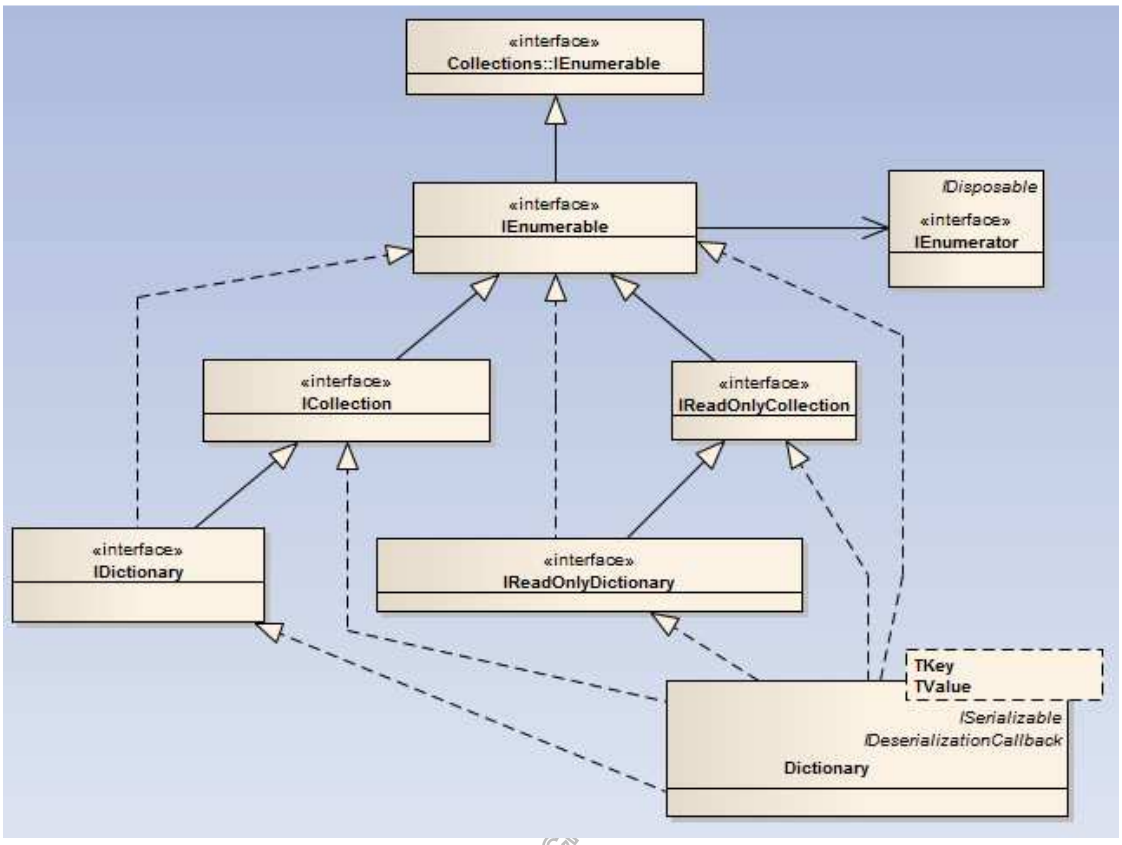
```
        conjuntoOrdenado.Add(a6)
        conjuntoOrdenado.Add(a2) ' Duplicado. No se agrega
        conjuntoOrdenado.Add(a3) ' Duplicado. No se agrega
        conjuntoOrdenado.Add(a1) ' Duplicado. No se agrega
        conjuntoOrdenado.Add(a2) ' Duplicado. No se agrega


        Console.WriteLine(
             "Ejemplo de conjunto que no admite duplicados y esta desordenado")
        MostrarLista(conjuntoOrdenado)

        Console.WriteLine("Cantidad de elementos en el conjunto: " + _
             conjuntoOrdenado.Count.ToString)
        Console.ReadKey()
    End Sub

    Public Sub MostrarLista(conjunto As ISet(Of Alumno))
        For Each elemento As Alumno In conjunto
            Console.WriteLine(elemento)
        Next
    End Sub
End Module
```

Dictionary

Representa una colección de pares de clave / valor que se organizan sobre la base del código hash que se obtiene a partir del objeto que actúa como clave. Cada elemento del diccionario está representado internamente por una estructura KeyValuePair<TKey, TValue> o KeyValuePair(Of TKey, TValue) (C# o VB). Cada adición al diccionario consta de un valor y su clave asociada. La recuperación de un valor utilizando su clave es muy rápida, ya que la clase Dictionary<TKey, TValue> o Dictionary (Of TKey, TValue) (C# o VB) se implementa como una tabla hash. Para localizar una clave, Dictionary calcula el código hash de la clave y luego lo usa para encontrarla en la tabla hash. Este tipo de búsqueda por clave es muy rápida.

Es importante tener en cuenta que la velocidad de recuperación está directamente asociada a la calidad del algoritmo utilizado en el método GetHashCode del objeto que actúa como clave. Siempre que un objeto se utilice como clave en Dictionary<TKey, TValue> o Dictionary (Of TKey, TValue) (C# o VB), no debe cambiar en cualquier forma que afecte su valor hash. Cada clave de la colección debe ser única conforme al comparador de igualdad usado en el diccionario. Una clave no puede ser null o Nothing (C# o VB), pero un valor sí puede serlo el valor asociado a ella si es un tipo por referencia.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra una implementación de la clase:

La calidad del algoritmo de para determinar el código hash puede ser obtenido de más de una forma. Dictionary<TKey, TValue> o Dictionary (Of TKey, TValue) (C# o VB) requiere una implementación que especifique la igualdad para determinar si las claves lo son. Se puede especificar una implementación de la interfaz genérica IEqualityComparer<TKey, TValue> o IEqualityComparer (Of T) (C# o VB) mediante uno de los constructores que acepte una referencia de este tipo como parámetro y así establecer los métodos de comparación (Equals y GetHashCode). Si no se especifica una implementación de este tipo, se utiliza la implementación de igualdad genérica predeterminada por defecto (es decir, los métodos Equals y GetHashCode

rescritos de Object), el cual es la implementación por defecto de IEqualityComparer<TKey, TValue> o IEqualityComparer (Of T) (C# o VB). Si el tipo TKey implementa la interfaz genérica System.IEquatable<T> o System.IEquatable(Of T) (C# o VB), el comparador de igualdad predeterminado utiliza dicha implementación.

El siguiente ejemplo muestra el uso de la determinación de igualdad de claves por defecto.

Ejemplo

C#

```
using estudiantesT;

namespace diccionarioT
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear un diccionario con un máximo de 4 entradas
            IDictionary<Horario, Alumno> d1 = new Dictionary<Horario, Alumno>(4);
            IDictionary<Horario, Alumno> d2 = new Dictionary<Horario, Alumno>(4);

            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
            Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

            Horario h1 = new Horario(2, 1, 0, 2, 20);
            Horario h2 = new Horario(2, 2, 20, 3, 40);
            Horario h3 = new Horario(2, 3, 40, 5, 0);
            Horario h4 = new Horario(3, 2, 20, 3, 40);
            Horario h5 = new Horario(4, 3, 40, 5, 0);

            // Poblar el primer diccionario
            AgregaEntrada(d1, h1, a1);
            AgregaEntrada(d1, h2, a2);
            AgregaEntrada(d1, h3, a3);
            AgregaEntrada(d1, h1, a4);
            AgregaEntrada(d1, h4, a4);

            // Poblar el segundo diccionario
            AgregaEntrada(d2, h1, a2);
            AgregaEntrada(d2, h2, a3);
            AgregaEntrada(d2, h3, a4);
            AgregaEntrada(d2, h2, a5);
            AgregaEntrada(d2, h5, a6);

            Console.WriteLine("Número de elementos en el diccionario 1 = {0}",
              d1.Count);
```

```
            Console.WriteLine("¿Se encuentra en el diccionario 1 {0} ? {1}", h2,
              d1.ContainsKey(h2));
            Console.WriteLine("El horario se asignó a " + d1[h2]);

            Console.WriteLine("\nMostrando el contenido del diccionario 1");
            ImprimirDiccionario(d1);

            Console.WriteLine("\nEliminando la entrada de " + h2);
            d1.Remove(h2);

            Console.WriteLine("\nMostrando el contenido del diccionario 1");
            ImprimirDiccionario(d1);

            Console.WriteLine("\nMostrando las claves del diccionario 1");
            MostrarClaves(d1);
            Console.WriteLine("\nMostrando los valores del diccionario 1");
            MostrarValores(d1);

            Console.WriteLine("----------------------------------------------");

            Console.WriteLine("\nNúmero de elementos en el diccionario 2 = {0}",
              d2.Count);

            Console.WriteLine("¿Se encuentra en el diccionario 2 {0} ? {1}", h2,
              d2.ContainsKey(h2));
            Console.WriteLine("El horario se asignó a " + d2[h2]);

            Console.WriteLine("\nMostrando el contenido del diccionario 2");
            ImprimirDiccionario(d2);

            Console.WriteLine("\nEliminando la entrada de " + h2);
            d2.Remove(h2);

            Console.WriteLine("\nMostrando el contenido del diccionario 2");
            ImprimirDiccionario(d2);

            Console.WriteLine("\nMostrando las claves del diccionario 2");
            MostrarClaves(d2);
            Console.WriteLine("\nMostrando los valores del diccionario 2");
            MostrarValores(d2);

            Console.ReadKey();
        }

        public static void AgregaEntrada(IDictionary<Horario, Alumno> d,
              Horario clave, Alumno valor)
        {
            try
            {
                d.Add(clave, valor);
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("Error. No se agregó entrada: " + e.Message);
            }
```

```
        }

        public static void ImprimirDiccionario(IDictionary<Horario, Alumno> d)
        {
            foreach (KeyValuePair<Horario, Alumno> de in d)
            {
                Console.WriteLine("Clave: {0}. Valor:  {1}.", de.Key, de.Value);
            }
        }

        public static void MostrarClaves(IDictionary<Horario, Alumno> d)
        {
            Console.WriteLine("Claves del diccionario:");
            foreach (Object o in d.Keys)
                Console.WriteLine(o);
        }

        public static void MostrarValores(IDictionary<Horario, Alumno> d)
        {
            Console.WriteLine("Valores del diccionario:");
            foreach (Object o in d.Values)
                Console.WriteLine(o);
        }
    }
}
```

VB

```
Imports estudiantesT

Module Module1

    Sub Main()
        Dim d1 As IDictionary(Of Horario, Alumno) = _
             New Dictionary(Of Horario, Alumno)()
        Dim d2 As IDictionary(Of Horario, Alumno) = _
             New Dictionary(Of Horario, Alumno)()

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        Dim h1 As New Horario(2, 1, 0, 2, 20)
        Dim h2 As New Horario(2, 2, 20, 3, 40)
        Dim h3 As New Horario(2, 3, 40, 5, 0)
        Dim h4 As New Horario(3, 2, 20, 3, 40)
        Dim h5 As New Horario(4, 3, 40, 5, 0)

        ' Poblar el primer diccionario
        AgregaEntrada(d1, h1, a1)
        AgregaEntrada(d1, h2, a2)
        AgregaEntrada(d1, h3, a3)
        AgregaEntrada(d1, h1, a4)
```

```
        AgregaEntrada(d1, h4, a4)

        ' Poblar el segundo diccionario
        AgregaEntrada(d2, h1, a2)
        AgregaEntrada(d2, h2, a3)
        AgregaEntrada(d2, h3, a4)
        AgregaEntrada(d2, h2, a5)
        AgregaEntrada(d2, h5, a6)

        Console.WriteLine("Número de elementos en el diccionario 1 = {0}", d1.Count)

        Console.WriteLine("¿Se encuentra en el diccionario 1 {0} ? {1}", h2,
             d1.ContainsKey(h2))
        Console.WriteLine("El horario se asignó a " + d1(h2).ToString)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 1")
        ImprimirDiccionario(d1)

        Console.WriteLine(vbNewLine + "Eliminando la entrada de " + h2.ToString)
        d1.Remove(h2)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 1")
        ImprimirDiccionario(d1)

        Console.WriteLine(vbNewLine + "Mostrando las claves del diccionario 1")
        MostrarClaves(d1)
        Console.WriteLine(vbNewLine + "Mostrando los valores del diccionario 1")
        MostrarValores(d1)

        Console.WriteLine("----------------------------------------")

        Console.WriteLine(vbNewLine + _
             "Número de elementos en el diccionario 2 = {0}", d2.Count)

        Console.WriteLine("¿Se encuentra en el diccionario 2 {0} ? {1}", _
             h2, d2.ContainsKey(h2))
        Console.WriteLine("El horario se asignó a " + d2(h2).ToString)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 2")
        ImprimirDiccionario(d2)

        Console.WriteLine(vbNewLine + "Eliminando la entrada de " + h2.ToString)
        d2.Remove(h2)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 2")
        ImprimirDiccionario(d2)

        Console.WriteLine(vbNewLine + "Mostrando las claves del diccionario 2")
        MostrarClaves(d2)
        Console.WriteLine(vbNewLine + "Mostrando los valores del diccionario 2")
        MostrarValores(d2)

        Console.ReadKey()
    End Sub
```

```
    Public Sub AgregaEntrada(d As IDictionary(Of Horario, Alumno), _
       clave As Horario, valor As Alumno)
        Try
            d.Add(clave, valor)
        Catch e As ArgumentException
            Console.WriteLine("Error. No se agregó entrada: " + e.Message)
        End Try
    End Sub

    Public Sub ImprimirDiccionario(d As IDictionary(Of Horario, Alumno))
        For Each de As KeyValuePair(Of Horario, Alumno) In d
            Console.WriteLine("Clave: {0}. Valor:  {1}.", de.Key, de.Value)
        Next
    End Sub

    Public Sub MostrarClaves(d As IDictionary(Of Horario, Alumno))
        Console.WriteLine("Claves del diccionario:")
        For Each o As Object In d.Keys
            Console.WriteLine(o)
        Next
    End Sub

    Public Sub MostrarValores(d As IDictionary(Of Horario, Alumno))
        Console.WriteLine("Valores del diccionario:")
        For Each o As Object In d.Values
            Console.WriteLine(o)
        Next
    End Sub
End Module
```


Sin embargo, usar IEqualityComparer permite una gran flexibilidad al momento de decidir como se deberán realizar las comparaciones, ya que una clase de este tipo puede determinar el criterio de comparación. Posteriormente se explicará el uso de esta clase para realizar comparaciones.

SortedList

La clase genérica SortedList<TKey, TValue> o SortedList(Of TKey, TValue) (C# o VB) es un árbol de búsqueda binaria. En este aspecto, es similar a la clase genérica SortedDictionary<TKey, TValue> o SortedDictionary(Of TKey, TValue) (C# o VB). Las dos clases tienen modelos de objetos similares y ambas recuperan elementos rápidamente. En lo que se diferencian las dos clases es en el uso de memoria y en la velocidad de inserción y eliminación:

- SortedList utiliza menos memoria que SortedDictionary.
- Las operaciones de inserción y eliminación de SortedDictionary para los datos no ordenados son mucho más rápidas.
- Si la lista sepuebla de una sola vez de datos que ya se encuentran ordenados, la colección SortedList es más rápida que SortedDictionary.

Otra diferencia entre las clases SortedDictionary y SortedList es que SortedList admite la recuperación indizada eficaz de claves y valores mediante las colecciones que devuelven las propiedades Keys y Values. Cuando se obtiene acceso a las propiedades, no es necesario volver a

generar las listas puesto que éstas únicamente son contenedores para los vectores internos de claves y valores.

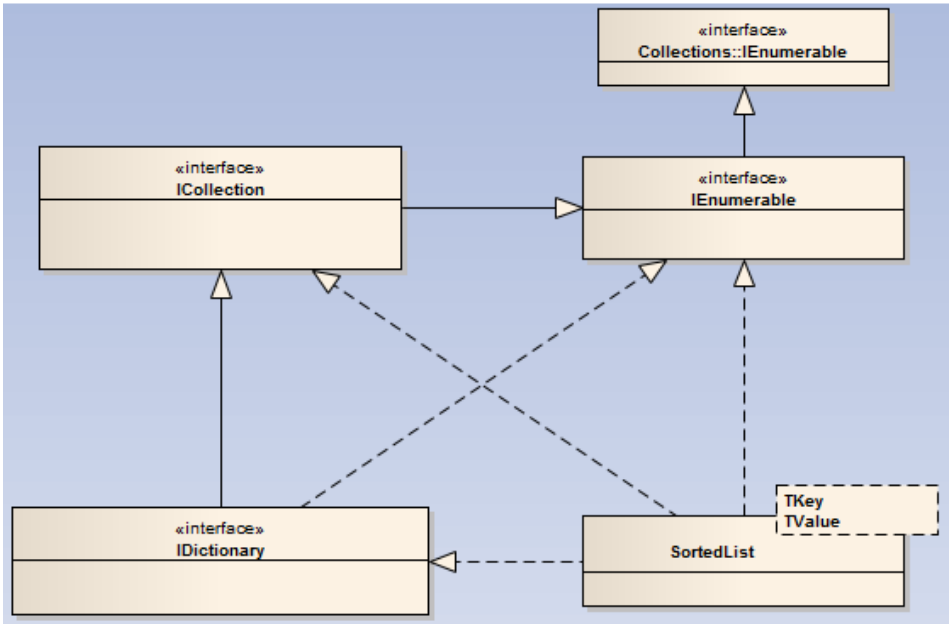
SortedList se implementa como un vector que contiene pares clave / valor, ordenado conforme la clave. Cada elemento se puede recuperar como un objeto del tipo KeyValuePair*. Esta es una diferencia importante respecto de la implementación no genérica de la clase en donde los elementos se pueden recuperar con los métodos GetKey(), GetByIndex() basándose en un índice recibido como argumento*.

Los objetos usados como claves deben permanecer inmutables mientras se utilicen como tales en un SortedList. Todas las claves de una colección de este tipo deben ser únicas. Una clave no puede ser una referencia `null` o `Nothing` (C# o VB), pero un valor sí puede serlo si es un tipo por referencia.

SortedList requiere la implementación de un comparador para ordenar y realizar comparaciones. Esto implica que la clave debe implementar mínimamente una interfaz del tipo IComparable<T> o IComparable(Of T) (C# o VB) para que se invoque al método CompareTo al momento de ordenar las claves, el cual es el comportamiento por defecto si no se provee en el constructor un objeto del tipo IComparer<T> o IComparer(Of T) (C# o VB) cuando se crea la instancia. Posteriormente se explicará el uso de comparadores para el ordenamiento de colecciones.

La capacidad de una colección SortedList es el número de elementos que dicha SortedList puede contener. Cuando se agregan elementos a una colección SortedList, la capacidad aumenta automáticamente según sea necesario mediante la reasignación del vector interno. La capacidad se puede disminuir si se llama al método TrimExcess o si se establece explícitamente la propiedad Capacity a un determinado valor. Al disminuir la capacidad se reasigna memoria y se copian todos los elementos de la colección SortedList.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra una implementación de la clase:

Ejemplo

C#

```
using estudiantesT;

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

```
            // Crea e inicializa un nuevo SortedList.
            SortedList<Horario, Alumno> listaOrdenada =
                    new SortedList<Horario, Alumno>();

            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
            Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

            Horario h1 = new Horario(2, 1, 0, 2, 20);
            Horario h2 = new Horario(2, 2, 20, 3, 40);
            Horario h3 = new Horario(2, 3, 40, 5, 0);
            Horario h4 = new Horario(3, 2, 20, 3, 40);
            Horario h5 = new Horario(4, 3, 40, 5, 0);

            listaOrdenada.Add(h1, a1);
            listaOrdenada.Add(h2, a2);
            listaOrdenada.Add(h3, a3);
            listaOrdenada.Add(h4, a4);
            listaOrdenada.Add(h5, a5);
            try
            {
                Console.WriteLine("Intentando agregar una clave que existe: " + h5);
                listaOrdenada.Add(h5, a6);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }

            // Mostrar propiedades y valores del SortedList.
            Console.WriteLine("Lista ordenada");
            Console.WriteLine("  Count:    {0}", listaOrdenada.Count);
            Console.WriteLine("  Capacity: {0}", listaOrdenada.Capacity);
            Console.WriteLine("  Claves y Valores:");
            ImprimirClavesYValores(listaOrdenada);

            Console.ReadKey();
        }

        public static void ImprimirClavesYValores(SortedList<Horario, Alumno> lista)
        {
            Console.WriteLine("\t-Clave-\t\t\t-Valor-");

            foreach (KeyValuePair<Horario, Alumno> kvp in lista)
            {
                Console.WriteLine("\t{0}:\t{1}", kvp.Key, kvp.Value);
            }
            Console.WriteLine();
        }
    }
}
```

VB

```
Imports estudiantesT

Module Module1

    Sub Main()
        ' Crea e inicializa un nuevo SortedList.
        Dim listaOrdenada As New SortedList(Of Horario, Alumno)

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        Dim h1 As New Horario(2, 1, 0, 2, 20)
        Dim h2 As New Horario(2, 2, 20, 3, 40)
        Dim h3 As New Horario(2, 3, 40, 5, 0)
        Dim h4 As New Horario(3, 2, 20, 3, 40)
        Dim h5 As New Horario(4, 3, 40, 5, 0)

        listaOrdenada.Add(h1, a1)
        listaOrdenada.Add(h2, a2)
        listaOrdenada.Add(h3, a3)
        listaOrdenada.Add(h4, a4)
        listaOrdenada.Add(h5, a5)
        Try
            Console.WriteLine("Intentando agregar una clave que existe: " _
               + h5.ToString)
            listaOrdenada.Add(h5, a6)
        Catch e As Exception
            Console.WriteLine(e.Message)
        End Try


        ' Mostrar propiedades y valores del SortedList.
        Console.WriteLine("Lista ordenada")
        Console.WriteLine("  Count:    {0}", listaOrdenada.Count)
        Console.WriteLine("  Capacity: {0}", listaOrdenada.Capacity)
        Console.WriteLine("  Claves y Valores:")
        ImprimirClavesYValores(listaOrdenada)

        Console.ReadKey()
    End Sub

    Public Sub ImprimirClavesYValores(lista As SortedList(Of Horario, Alumno))
        Console.WriteLine(vbTab + "-Clave-" + vbTab + vbTab + vbTab + "-Valor-")

        For Each kvp As KeyValuePair(Of Horario, Alumno) In lista
            Console.WriteLine(vbTab + "{0}:" + vbTab + "{1}", kvp.Key, kvp.Value)
        Next
        Console.WriteLine()
    End Sub
```

```
End Module
```

SortedDictionary

La clase genérica SortedDictionary<TKey, TValue> o SortedDictionary(Of TKey, TValue) (C# o VB) es un árbol de búsqueda binaria. En este aspecto, es similar a la clase genérica SortedList<TKey, TValue> o SortedList(Of TKey, TValue) (C# o VB) pero al ser un árbol el algoritmo implementado la convierte en la candidata ideal para ir ingresando elementos que inicialmente se encuentran desordenados y que se vayan ordenando adecuadamente a medida que se agregan a la colección. Las dos clases tienen modelos de objetos similares y ambas recuperan rápidamente. En lo que se diferencian las dos clases es en el uso de memoria y en la velocidad de inserción y eliminación:

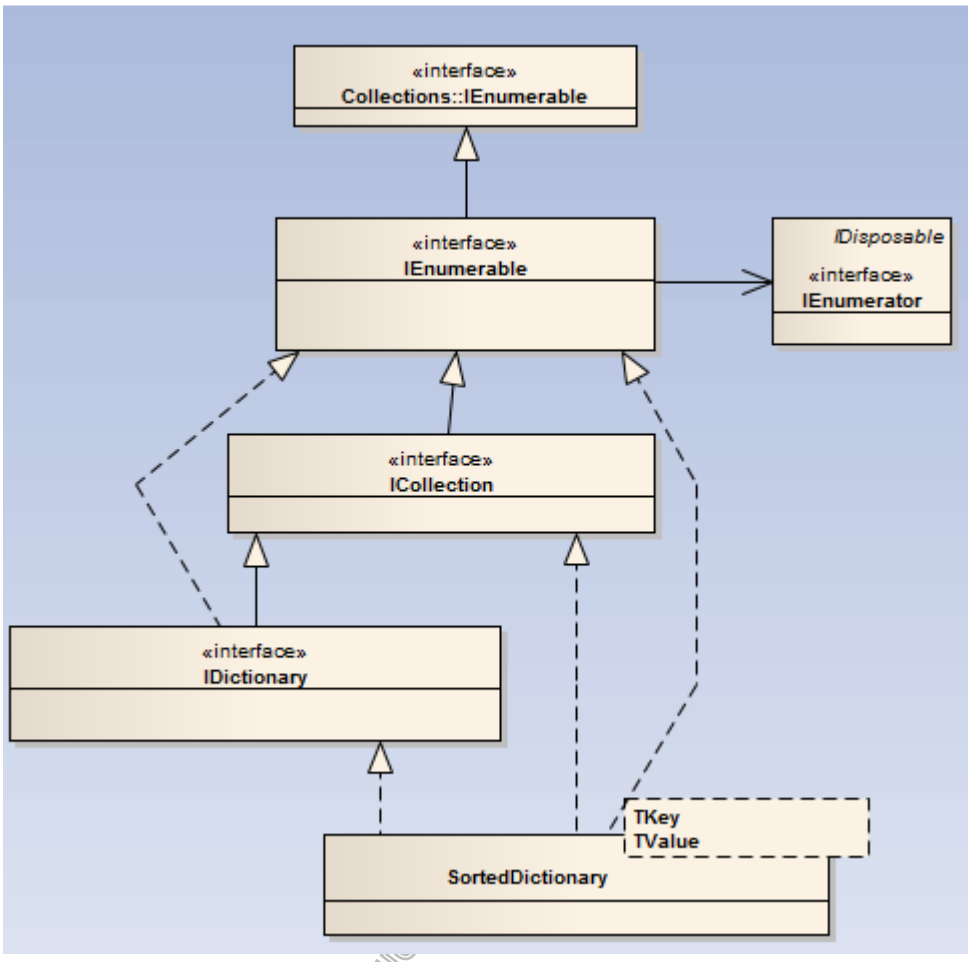
- SortedList<TKey, TValue> o SortedList(Of TKey, TValue) (C# o VB) utiliza menos memoria que SortedDictionary<TKey, TValue> o SortedDictionary(Of TKey, TValue) (C# o VB).
- Las operaciones de inserción y eliminación de SortedDictionary<TKey, TValue> o SortedDictionary(Of TKey, TValue) (C# o VB) para los datos no ordenados son mucho más rápidas.
- Si la lista se rellena de una sola vez de datos ordenados, la colección SortedList<TKey, TValue> o SortedList(Of TKey, TValue) (C# o VB) es más rápida que SortedDictionary<TKey, TValue> o SortedDictionary(Of TKey, TValue) (C# o VB).

Cada par clave/valor se puede recuperar como una estructura KeyValuePair<TKey, TValue> o KeyValuePair(Of TKey, TValue) (C# o VB) o como DictionaryEntry a través de la interfaz IDictionary no genérica.

Las claves deben permanecer inmutables mientras se utilicen como claves en la colección SortedDictionary. Todas las claves de una colección de este tipo deben ser únicas. Una clave no puede ser una referencia `null` o `Nothing` (C# o VB), pero un valor sí puede serlo si es un tipo por referencia.

SortedDictionary requiere la implementación de un comparador para ordenar y realizar comparaciones. Esto implica que la clave debe implementar mínimamente una interfaz del tipo IComparable<T> o IComparable(Of T) (C# o VB) para que se invoque al método CompareTo al momento de ordenar las claves, el cual es el comportamiento por defecto si no se provee en el constructor un objeto del tipo IComparer<T> o IComparer(Of T) (C# o VB)cuando se crea la instancia. Posteriormente se explicará el uso de comparadores para el ordenamiento de colecciones.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra una implementación de la clase:

Ejemplo

C#

```
using estudiantesT;

namespace diccionarioOrdenadoT
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear un diccionario ordenado
            IDictionary<Horario, Alumno> d1 =
              new SortedDictionary<Horario, Alumno>();
            IDictionary<Horario, Alumno> d2 =
              new SortedDictionary<Horario, Alumno>();
```

```
Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

Horario h1 = new Horario(2, 1, 0, 2, 20);
Horario h2 = new Horario(2, 2, 20, 3, 40);
Horario h3 = new Horario(2, 3, 40, 5, 0);
Horario h4 = new Horario(3, 2, 20, 3, 40);
Horario h5 = new Horario(4, 3, 40, 5, 0);

// Poblar el primer diccionario
AgregaEntrada(d1, h1, a1);
AgregaEntrada(d1, h2, a2);
AgregaEntrada(d1, h3, a3);
AgregaEntrada(d1, h1, a4);
AgregaEntrada(d1, h4, a4);

// Poblar el segundo diccionario
AgregaEntrada(d2, h1, a2);
AgregaEntrada(d2, h2, a3);
AgregaEntrada(d2, h3, a4);
AgregaEntrada(d2, h2, a5);
AgregaEntrada(d2, h5, a6);

Console.WriteLine("Número de elementos en el diccionario 1 = {0}",
  d1.Count);

Console.WriteLine("¿Se encuentra en el diccionario 1 {0} ? {1}", h2,
  d1.ContainsKey(h2));
Console.WriteLine("El horario se asignó a " + d1[h2]);

Console.WriteLine("\nMostrando el contenido del diccionario 1");
ImprimirDiccionario(d1);

Console.WriteLine("\nEliminando la entrada de " + h2);
d1.Remove(h2);

Console.WriteLine("\nMostrando el contenido del diccionario 1");
ImprimirDiccionario(d1);

Console.WriteLine("\nMostrando las claves del diccionario 1");
MostrarClaves(d1);
Console.WriteLine("\nMostrando los valores del diccionario 1");
MostrarValores(d1);

Console.WriteLine("-----------------------------------------------");

Console.WriteLine("\nNúmero de elementos en el diccionario 2 = {0}",
  d2.Count);
```

```
            Console.WriteLine("¿Se encuentra en el diccionario 2 {0} ? {1}", h2,
              d2.ContainsKey(h2));
            Console.WriteLine("El horario se asignó a " + d2[h2]);

            Console.WriteLine("\nMostrando el contenido del diccionario 2");
            ImprimirDiccionario(d2);

            Console.WriteLine("\nEliminando la entrada de " + h2);
            d2.Remove(h2);

            Console.WriteLine("\nMostrando el contenido del diccionario 2");
            ImprimirDiccionario(d2);

            Console.WriteLine("\nMostrando las claves del diccionario 2");
            MostrarClaves(d2);
            Console.WriteLine("\nMostrando los valores del diccionario 2");
            MostrarValores(d2);

            Console.ReadKey();
        }

        public static void AgregaEntrada(IDictionary<Horario, Alumno> d,
              Horario clave, Alumno valor)
        {
            try
            {
                d.Add(clave, valor);
            }
            catch (ArgumentException e)
            {
                Console.WriteLine("Error. No se agregó entrada: " + e.Message);
            }
        }

        public static void ImprimirDiccionario(IDictionary<Horario, Alumno> d)
        {
            foreach (KeyValuePair<Horario, Alumno> de in d)
            {
                Console.WriteLine("Clave: {0}. Valor:  {1}.", de.Key, de.Value);
            }
        }

        public static void MostrarClaves(IDictionary<Horario, Alumno> d)
        {
            Console.WriteLine("Claves del diccionario:");
            foreach (Object o in d.Keys)
                Console.WriteLine(o);
        }

        public static void MostrarValores(IDictionary<Horario, Alumno> d)
        {
            Console.WriteLine("Valores del diccionario:");
            foreach (Object o in d.Values)
                Console.WriteLine(o);
        }
```

```
    }
}
```

VB

```
Imports estudiantesT

Module Module1

    Sub Main()
        ' Crear un diccionario ordenado
        Dim d1 As IDictionary(Of Horario, Alumno) = _
             New SortedDictionary(Of Horario, Alumno)()
        Dim d2 As IDictionary(Of Horario, Alumno) = _
             New SortedDictionary(Of Horario, Alumno)()

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 As New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        Dim h1 As New Horario(2, 1, 0, 2, 20)
        Dim h2 As New Horario(2, 2, 20, 3, 40)
        Dim h3 As New Horario(2, 3, 40, 5, 0)
        Dim h4 As New Horario(3, 2, 20, 3, 40)
        Dim h5 As New Horario(4, 3, 40, 5, 0)

        ' Poblar el primer diccionario
        AgregaEntrada(d1, h1, a1)
        AgregaEntrada(d1, h2, a2)
        AgregaEntrada(d1, h3, a3)
        AgregaEntrada(d1, h1, a4)
        AgregaEntrada(d1, h4, a4)

        ' Poblar el segundo diccionario
        AgregaEntrada(d2, h1, a2)
        AgregaEntrada(d2, h2, a3)
        AgregaEntrada(d2, h3, a4)
        AgregaEntrada(d2, h2, a5)
        AgregaEntrada(d2, h5, a6)

        Console.WriteLine("Número de elementos en el diccionario 1 = {0}", d1.Count)

        Console.WriteLine("¿Se encuentra en el diccionario 1 {0} ? {1}",
             h2, d1.ContainsKey(h2))
        Console.WriteLine("El horario se asignó a " + d1(h2).ToString)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 1")
        ImprimirDiccionario(d1)

        Console.WriteLine(vbNewLine + "Eliminando la entrada de " + h2.ToString)
        d1.Remove(h2)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 1")
```

```
        ImprimirDiccionario(d1)

        Console.WriteLine(vbNewLine + "Mostrando las claves del diccionario 1")
        MostrarClaves(d1)
        Console.WriteLine(vbNewLine + "Mostrando los valores del diccionario 1")
        MostrarValores(d1)

        Console.WriteLine("-----------------------------------------------")

        Console.WriteLine(vbNewLine + _
              "Número de elementos en el diccionario 2 = {0}", d2.Count)

        Console.WriteLine("¿Se encuentra en el diccionario 2 {0} ? {1}",
              h2, d2.ContainsKey(h2))
        Console.WriteLine("El horario se asignó a " + d2(h2).ToString)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 2")
        ImprimirDiccionario(d2)

        Console.WriteLine(vbNewLine + "Eliminando la entrada de " + h2.ToString)
        d2.Remove(h2)

        Console.WriteLine(vbNewLine + "Mostrando el contenido del diccionario 2")
        ImprimirDiccionario(d2)

        Console.WriteLine(vbNewLine + "Mostrando las claves del diccionario 2")
        MostrarClaves(d2)
        Console.WriteLine(vbNewLine + "Mostrando los valores del diccionario 2")
        MostrarValores(d2)

        Console.ReadKey()
    End Sub

    Public Sub AgregaEntrada(d As IDictionary(Of Horario, Alumno), _
       clave As Horario, valor As Alumno)
        Try
            d.Add(clave, valor)
        Catch e As ArgumentException
            Console.WriteLine("Error. No se agregó entrada: " + e.Message)
        End Try
    End Sub

    Public Sub ImprimirDiccionario(d As IDictionary(Of Horario, Alumno))
        For Each de As KeyValuePair(Of Horario, Alumno) In d
            Console.WriteLine("Clave: {0}. Valor:  {1}.", de.Key, de.Value)
        Next
    End Sub

    Public Sub MostrarClaves(d As IDictionary(Of Horario, Alumno))
        Console.WriteLine("Claves del diccionario:")
        For Each o As Object In d.Keys
            Console.WriteLine(o)
        Next
    End Sub
```

```
    Public Sub MostrarValores(d As IDictionary(Of Horario, Alumno))
        Console.WriteLine("Valores del diccionario:")
        For Each o As Object In d.Values
            Console.WriteLine(o)
        Next
    End Sub
End Module
```

Comparer

La clase Comparer<T> o Comparer (Of T) (C# o VB) implementa la interfaz IComparer<T> o IComparer(Of T) (C# o VB) y brinda la posibilidad de crear un criterio de comparación personalizado implementándolo por rescritura en el método Compare, el cual retorna si es igual a, menor o mayor que el primer objeto que se recibe como argumento respecto del segundo.

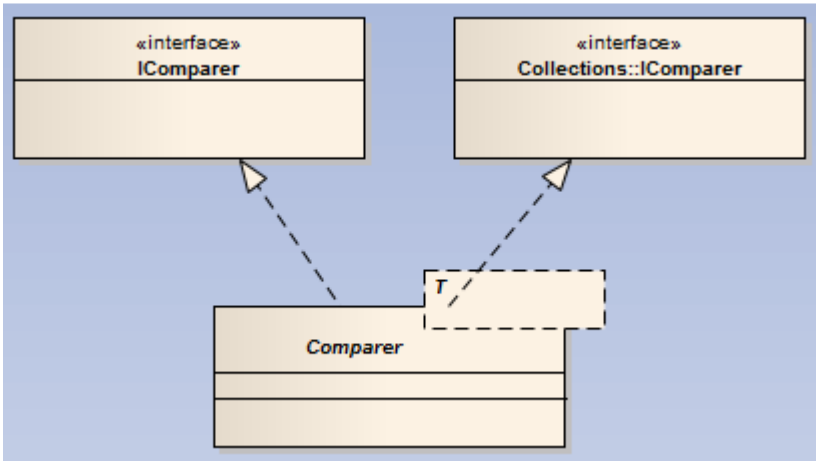
Se puede derivar de esta clase para proporcionar una implementación personalizada de la interfaz IComparer<T> con el fin de utilizarla con las clases de colección, como las clases genéricas SortedList<TKey, TValue> o SortedList(Of TKey, TValue) (C# oVB) y SortedDictionary<TKey, TValue> o SortedDictionary(Of TKey, TValue) (C# oVB).

La diferencia entre derivar de la clase Comparer<T> o Comparer(Of T) (C# oVB) e implementar la interfaz System.IComparable<T> o System.IComparable(Of T) (C# oVB) es la siguiente:

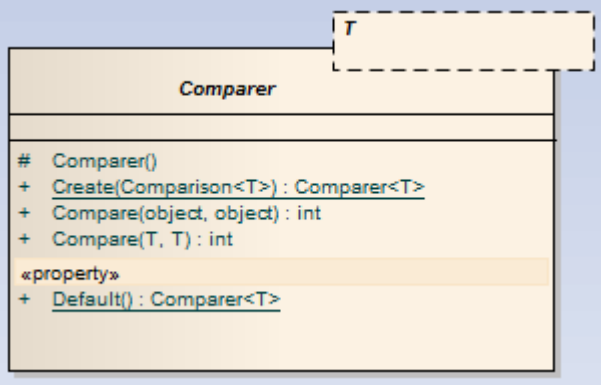
- Para especificar cómo se debe comparar dos objetos de forma predeterminada, implementar la interfaz System.IComparable en la clase que será sujata a la comparación. Esto garantiza que las operaciones de ordenación utilizarán el código de comparación predeterminado que se proporcionó.
- Para definir un comparador para utilizarlo en lugar del comparador predeterminado, derivar de la clase Comparer<T> o Comparer(Of T) (C# oVB). A continuación, se puede utilizar este comparador en operaciones de ordenación que tomen un comparador como parámetro. Particularmente se puede especificar como argumento en las colecciones que lo soporten en el constructor y determinar así el criterio de comparación por defecto para dicha instancia.

El objeto que devuelve la propiedad Default utiliza la interfaz genérica System.IComparable<T> o System.IComparable(Of T) (C# o VB) para comparar dos objetos. Si el tipo T no implementa la interfaz genérica System.IComparable<T> o IComparable(Of T) (C# o VB), la propiedad Default retorna un objeto del tipo Comparer<T> que utiliza la interfaz System.IComparable que define el criterio de comparación.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra una implementación de la clase:



Ejemplo

C#

```
using estudiantesT;

namespace comparadorT
{
    class AlumnosPorEdad: Comparer<Alumno>
    {
        public override int Compare(Alumno obj1, Alumno obj2)
        {
            return obj1.Edad - obj2.Edad;
        }
    }
}

using estudiantesT;

namespace comparadorT
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Alumno> alumnos = new List<Alumno>();

            Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
            Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
            Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
            Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
            Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);

            alumnos.Add(a1);
            alumnos.Add(a2);
            alumnos.Add(a3);
            alumnos.Add(a4);
            alumnos.Add(a5);

            Console.WriteLine("Usando el comparador por edad: ");
            // Ordenar usando el comparador
            alumnos.Sort(new AlumnosPorEdad());

            foreach (Alumno alumno in alumnos)
            {
                Console.Write(alumno);
                Console.WriteLine("\tEdad: " + alumno.Edad);
            }

            Comparer<Alumno> comparadorPorDefecto = Comparer<Alumno>.Default;

            Console.WriteLine("\nUsando la comparación por defecto: ");
            // Ordenar usando el comparador por defecto
            alumnos.Sort(comparadorPorDefecto);

            foreach (Alumno alumno in alumnos)
            {
                Console.Write(alumno);
                Console.WriteLine("\tEdad: " + alumno.Edad);
            }

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


VB

```
Imports estudiantesT

Public Class AlumnosPorEdad
    Inherits Comparer(Of Alumno)

    Public Overrides Function Compare(obj1 As Alumno, obj2 As Alumno) As Integer
        Return obj1.Edad - obj2.Edad
    End Function
```

```
End Class

Imports estudiantesT

Module Module1

    Sub Main()
        Dim alumnos As New List(Of Alumno)

        Dim a1 As New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 As New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 As New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 As New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 As New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)

        alumnos.Add(a1)
        alumnos.Add(a2)
        alumnos.Add(a3)
        alumnos.Add(a4)
        alumnos.Add(a5)

        Console.WriteLine("Usando el comparador por edad: ")
        ' Ordenar usando el comparador
        alumnos.Sort(New AlumnosPorEdad())

        For Each alumno As Alumno In alumnos
            Console.Write(alumno)
            Console.WriteLine(vbTab + "Edad: " + alumno.Edad.ToString)
        Next

        Dim comparadorPorDefecto As Comparer(Of Alumno) = _
             Comparer(Of Alumno).Default

        Console.WriteLine(vbNewLine + "Usando la comparación por defecto: ")
        ' Ordenar usando el comparador por defecto
        alumnos.Sort(comparadorPorDefecto)

        For Each alumno As Alumno In alumnos
            Console.Write(alumno)
            Console.WriteLine(vbTab + "Edad: " + alumno.Edad.ToString)
        Next

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


EqualityComparer

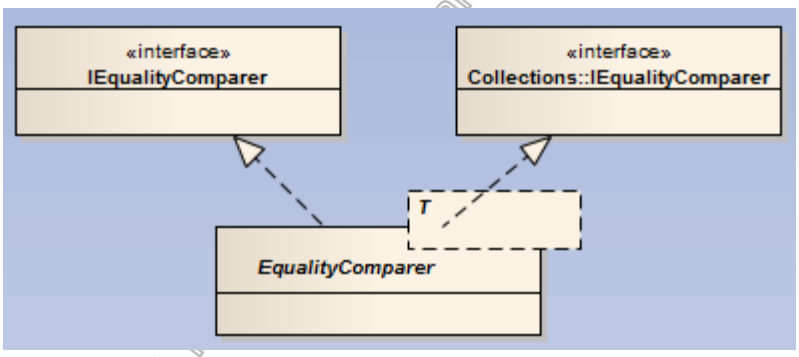
La clase EqualityComparer <T> o EqualityComparer (Of T) (C# o VB) implementa la interfaz IEqualityComparer <T> o IEqualityComparer (Of T) (C# o VB) y brinda la posibilidad de crear un criterio de comparación personalizado implementándolo por rescritura en los métodos Equals y GetHashCode, los cuales tienen un criterio de implementación y comportamiento igual al ya explicado para la clase Object. La diferencia fundamental radica en poder crear criterios diferentes a partir de rescribir los métodos que brinda la clase.

Se puede derivar de esta clase para proporcionar una implementación personalizada de la interfaz genérica IEqualityComparer<T> para utilizarla con las clases de colecciones como la clase genérica Dictionary<TKey, TValue> o Dictionary(Of TKey, TValue) (C# o VB) o con métodos como List<T>.Sort o List(Of T).Sort (C# o VB).

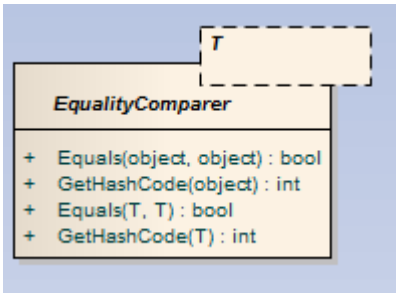
La propiedad Default comprueba si el tipo T implementa la interfaz genérica System.IEquatable<T> o System.IEquatable(Of T) (C# o VB) y, en tal caso, devuelve un objeto del tipo EqualityComparer<T> o EqualityComparer(Of T) (C# o VB) que contiene la implementación del método IEquatable<T>.Equals o IEquatable(Of T).Equals (C# o VB). De lo contrario, retorna EqualityComparer<T> o EqualityComparer(Of T) (C# o VB), que ha proporcionado el parámetro genérico T.

Se recomienda que se derive de la clase EqualityComparer en lugar de implementar la interfaz IEqualityComparer, porque las detrminaciones de objetos iguales se realizan a través del método IEquatable<T>.Equals en lugar del método Object.Equals. Esto es coherente con los métodos Contains, IndexOf, Remove y LastIndexOf de la clase genérica Dictionary como así también para otras colecciones genéricas.

El siguiente diagrama UML representa la clase con las interfaces que implementa:



El siguiente diagrama UML muestra una implementación de la clase:



Ejemplo

C#

```
using estudiantesT;

namespace comparadorIgualdadT
{
    class HorarioIgualDia : EqualityComparer<Horario>
    {

        public override bool Equals(Horario x, Horario y)
        {
            if (x.Dia == y.Dia) return true;
            return false;
        }

        public override int GetHashCode(Horario obj)
        {
            return obj.Dia;
        }
    }
}

using estudiantesT;

namespace comparadorIgualdadT
{
    class HorarioIgualHora : EqualityComparer<Horario>
    {
        public override bool Equals(Horario x, Horario y)
        {
            if (x.HoraComienzo == y.HoraComienzo && x.HoraFin == y.HoraFin
              && x.MinutosComienzo == y.MinutosComienzo &&
              x.MinutosFin == y.MinutosFin)
                 return true;
            return false;
        }

        public override int GetHashCode(Horario obj)
        {
            return obj.HoraComienzo ^ obj.MinutosComienzo ^ obj.HoraFin ^
                    obj.MinutosFin;
        }
    }
}

using estudiantesT;

namespace comparadorIgualdadT
{
    class Program
    {
        static Dictionary<Horario, Alumno> horarios;
```

```
static void Main(string[] args)
{
    Alumno a1 = new Alumno("Juan", "Adrián", "Lozano", 22, 12345);
    Alumno a2 = new Alumno("Fabio", "Miguel", "Sivori", 21, 11111);
    Alumno a3 = new Alumno("Eduardo", "Dario", "Masche", 20, 22222);
    Alumno a4 = new Alumno("Román", "Federico", "Giuta", 24, 66666);
    Alumno a5 = new Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555);
    Alumno a6 = new Alumno("Pedro", "Alejo", "Farfán", 20, 23232);

    // Construir un diccionario donde el criterio para
    // comparar claves sea por la igualdad de los días
    HorarioIgualDia horariosDelMismoDia = new HorarioIgualDia();
    horarios = new Dictionary<Horario, Alumno>(horariosDelMismoDia);

    Console.WriteLine("Horarios en el mismo día");
    Horario h1 = new Horario(2, 1, 0, 2, 20);
    Horario h2 = new Horario(2, 2, 20, 3, 40);
    Horario h3 = new Horario(2, 3, 40, 5, 0);
    Horario h4 = new Horario(3, 2, 20, 3, 40);
    Horario h5 = new Horario(4, 3, 40, 5, 0);

    AgregarHorario(h1, a1);
    AgregarHorario(h2, a2);
    AgregarHorario(h3, a3);
    AgregarHorario(h4, a4);
    AgregarHorario(h5, a5);

    Console.WriteLine("Cantidad de objetos en el diccionario: " +
            horarios.Count);

    Console.WriteLine();

    // Construir un diccionario donde el criterio para
    // comparar claves sea por la igualdad de hora y minutos
    HorarioIgualHora horariosConLaMismaHora = new HorarioIgualHora();
    horarios = new Dictionary<Horario, Alumno>(horariosConLaMismaHora);

    Console.WriteLine("Horarios con la misma hora");
    Horario h6 = new Horario(15, 2, 20, 3, 30);
    Horario h7 = new Horario(18, 2, 20, 3, 30);
    Horario h8 = new Horario(22, 2, 20, 3, 30);
    Horario h9 = new Horario(18, 4, 20, 5, 30);

    AgregarHorario(h6, a1);
    AgregarHorario(h7, a2);
    AgregarHorario(h8, a3);
    AgregarHorario(h9, a4);

    Console.WriteLine("Cantidad de objetos en el diccionario: " +
      horarios.Count);

    Console.ReadKey();
}

public static void AgregarHorario(Horario horario, Alumno alumno)
```

```
        {
            try
            {
                horarios.Add(horario, alumno);
                // El HashCode del objeto que es clave no es el que se utiliza
                // para realizar la comparación de la clave, sino el de
                //EqualityComparer
                Console.WriteLine(
                    "Agregado {0}, Horario {1} Cantidad = {2}, HashCode = {3}",
                    alumno, horario, horarios.Count.ToString(),
                    horario.GetHashCode());
            }
            catch (ArgumentException)
            {
                Console.WriteLine(
                    "Un horario igual a {0} ya se encuentra en la colección",
                    horario);
                Console.WriteLine("No se agrega el horario para el alumno {0} .",
                    alumno);
            }
        }
    }
}
```

VB

```
Imports estudiantesT

Public Class HorarioIgualDia
    Inherits EqualityComparer(Of Horario)

    Public Overloads Overrides Function Equals(x As estudiantesT.Horario, _
       y As estudiantesT.Horario) As Boolean
        If x.Dia = y.Dia Then Return True
        Return False
    End Function

    Public Overloads Overrides Function GetHashCode(obj As estudiantesT.Horario) _
       As Integer
        Return obj.Dia
    End Function
End Class

Imports estudiantesT

Public Class HorarioIgualHora
    Inherits EqualityComparer(Of Horario)

    Public Overloads Overrides Function Equals(_
       x As estudiantesT.Horario, y As estudiantesT.Horario) As Boolean
        If x.HoraComienzo = y.HoraComienzo AndAlso x.HoraFin = y.HoraFin AndAlso _
                x.MinutosComienzo = y.MinutosComienzo AndAlso _
                 x.MinutosFin = y.MinutosFin Then
            Return True
        End If
        Return False
```

```
    End Function

    Public Overloads Overrides Function GetHashCode(obj As estudiantesT.Horario) _
       As Integer
        Return obj.HoraComienzo Xor obj.MinutosComienzo Xor _
              obj.HoraFin Xor obj.MinutosFin
    End Function
End Class

Imports estudiantesT

Module Module1
    Dim horarios As Dictionary(Of Horario, [Alumno])

    Sub Main()
        Dim a1 = New Alumno("Juan", "Adrián", "Lozano", 22, 12345)
        Dim a2 = New Alumno("Fabio", "Miguel", "Sivori", 21, 11111)
        Dim a3 = New Alumno("Eduardo", "Dario", "Masche", 20, 22222)
        Dim a4 = New Alumno("Román", "Federico", "Giuta", 24, 66666)
        Dim a5 = New Alumno("Ignacio", "Fabián", "Insaurralde", 25, 55555)
        Dim a6 = New Alumno("Pedro", "Alejo", "Farfán", 20, 23232)

        ' Construir un diccionario donde el criterio para
        ' comparar claves sea por la igualdad de los días
        Dim horariosDelMismoDia As HorarioIgualDia = New HorarioIgualDia()
        horarios = New Dictionary(Of Horario, [Alumno])(horariosDelMismoDia)

        Console.WriteLine("Horarios en el mismo día")
        Dim h1 = New Horario(2, 1, 0, 2, 20)
        Dim h2 = New Horario(2, 2, 20, 3, 40)
        Dim h3 = New Horario(2, 3, 40, 5, 0)
        Dim h4 = New Horario(3, 2, 20, 3, 40)
        Dim h5 = New Horario(4, 3, 40, 5, 0)

        AgregarHorario(h1, a1)
        AgregarHorario(h2, a2)
        AgregarHorario(h3, a3)
        AgregarHorario(h4, a4)
        AgregarHorario(h5, a5)

        Console.WriteLine("Cantidad de objetos en el diccionario: " +
              horarios.Count.ToString)

        Console.WriteLine()

        ' Construir un diccionario donde el criterio para
        ' comparar claves sea por la igualdad de hora y minutos
        Dim horariosConLaMismaHora As HorarioIgualHora = New HorarioIgualHora()
        horarios = New Dictionary(Of Horario, [Alumno])(horariosConLaMismaHora)

        Console.WriteLine("Horarios con la misma hora")
        Dim h6 As New Horario(15, 2, 20, 3, 30)
        Dim h7 As New Horario(18, 2, 20, 3, 30)
        Dim h8 As New Horario(22, 2, 20, 3, 30)
        Dim h9 As New Horario(18, 4, 20, 5, 30)
```

```
        AgregarHorario(h6, a1)
        AgregarHorario(h7, a2)
        AgregarHorario(h8, a3)
        AgregarHorario(h9, a4)

        Console.WriteLine("Cantidad de objetos en el diccionario: " +
              horarios.Count.ToString)

        Console.ReadKey()
    End Sub

    Public Sub AgregarHorario(horario As Horario, alumno As Alumno)
        Try
            horarios.Add(horario, alumno)
            ' El HashCode del objeto que es clave no es el que se utiliza
            ' para realizar la comparación de la clave, sino el de EqualityComparer
            Console.WriteLine(
              "Agregado {0}, Horario {1} Cantidad = {2}, HashCode = {3}", _
                alumno, horario, horarios.Count.ToString(), horario.GetHashCode())
        Catch e As ArgumentException
            Console.WriteLine(
              "Un horario igual a {0} ya se encuentra en la colección", horario)
            Console.WriteLine(
              "No se agrega el horario para el alumno {0} .", alumno)
        End Try
    End Sub
End Module
```


Covarianza y contravarianza

La covarianza y la contravarianza determinan como se realizan las conversions de tipo para los parámetros utilizados en los genéricos. Las interfaces genéricas que se utilizan en las colecciones definen estas conversiones en sus declaraciones. Entre las que se fueron nombrando en este capítulo se pueden mencionar:

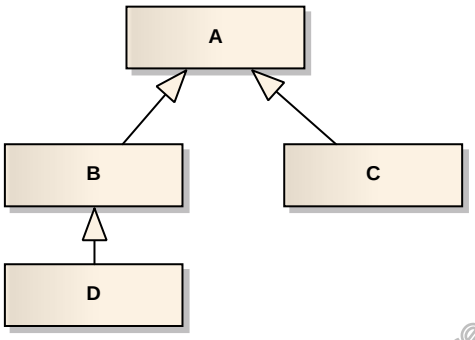
- IEnumerable (T es covariante)
- IEnumerator (T es covariante)
- IComparer (T es contravariante)
- IEqualityComparer (T es contravariante)
- IComparable (T es contravariante)

La covarianza permite cierta flexibilidad para trabajar con parámetros de tipos. Cuando se trabaja con una interfaz definida para ser covariante y se define un tipo como parámtero de la misma, si dicha interfaz es implementada en una clase, la misma permitirá como parámetro al definido y cualquiera que sea subtipo de él.

Es importante entender que la conversión de tipo la verifica el compilador por medio de la inferencia, es decir, el compilador “infiere” que los tipos de los parámetros son convertibles y

realiza las asignaciones especificadas. No se debe confundir esto con polimo0rfismo, el cual se realiza en tiempo de ejecución.

Para los ejemplos que se van a mostrar a continuación, tener presente las cadenas de herencia que muestra el siguiente diagrama:



El siguiente ejemplo ilustra el concepto. Notar que lo único que hace el compilador es verificar si el parámetro que se le pasa al genérico es convertible.

Ejemplo

C#

```
IEnumerable<String> strings = new List<String>();
IEnumerable<Object> objects = strings;
```

VB

```
Dim strings As IEnumerable(Of String) = New List(Of String)
Dim objects As IEnumerable(Of Object) = strings
```

Se pueden definir tanto interfaces como delegados para que sean covariantes. La única salvedad es especificar el parámetro con la palabra clave `out`.

Ejemplo

C#

```
namespace covarianza
{
    class Enumerador : IEnumerable<A>, IEnumerator<A>
    {
        private A[] vec;
        private int indice = -1;

        public Enumerador(A[] _vec)
```

```
{
    vec = new A[_vec.Length];
    for (int i = 0; i < _vec.Length; i++)
    {
        vec[i] = _vec[i];
    }
}

public IEnumerator<A> GetEnumerator()
{
    return (IEnumerator<A>)this;
}

System.Collections.IEnumerator
     System.Collections.IEnumerable.GetEnumerator()
{
    throw new NotImplementedException();
}


public A Current
{
    get
    {
        try
        {
            if (vec[indice] == null) throw new InvalidOperationException(
                   "No hay elementos");
            return vec[indice];
        }
        catch (IndexOutOfRangeException)
        {
            throw new InvalidOperationException(
                   "No hay más elementos almacenados");
        }
    }
}

public void Dispose()
{

}

object System.Collections.IEnumerator.Current
{
    get { throw new NotImplementedException(); }
}

public bool MoveNext()
{
    indice++;
    return (indice < vec.Length);
}

public void Reset()
```

```
        {
            indice = -1;
        }
    }
}

namespace covarianza
{
    class Program
    {
        static void Main(string[] args)
        {
            A[] vecA = { new A(1), new A(2), new A(3) };
            B[] vecB = { new B(4), new B(5), new B(6) };
            C[] vecC = { new C(7), new C(8), new C(9) };
            D[] vecD = { new D(10), new D(11), new D(12) };

            // Todas estas instacias son correctas
            // porque IEnumerable e IEnumerator son
            // covariantes respecto de su parámetro
            Enumerador listaDeA = new Enumerador(vecA);
            Enumerador listaDeB = new Enumerador(vecB);
            Enumerador listaDeC = new Enumerador(vecC);
            Enumerador listaDeD = new Enumerador(vecD);

            // Usar los enumeradores definidos por covarianza
            foreach (A _a in listaDeA)
            {
                Console.Write("Estoy en ");
                _a.MiMetodo();
                Console.WriteLine("El valor de Var1 es:  " + _a.Var1);
            }
            foreach (B _b in listaDeB)
            {
                Console.Write("Estoy en ");
                _b.MiMetodo();
                Console.WriteLine("El valor de Var1 es:  " + _b.Var1);
            }
            foreach (C _c in listaDeC)
            {
                Console.Write("Estoy en ");
                _c.MiMetodo();
                Console.WriteLine("El valor de Var1 es:  " + _c.Var1);
            }
            foreach (D _d in listaDeD)
            {
                Console.Write("Estoy en ");
                _d.MiMetodo();
                Console.WriteLine("El valor de Var1 es:  " + _d.Var1);
            }

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

VB

```
Public Class Enumerador
    Implements IEnumerable(Of A), IEnumerator(Of A)

    Private _vec() As A
    Private indice As Integer = -1

    Public Sub New(_vec() As A)
        ReDim Me._vec(_vec.Length - 1)
        For i As Integer = 0 To _vec.Length - 1
            Me._vec(i) = _vec(i)
        Next
    End Sub

    Public Function GetEnumerator() As System.Collections.Generic.IEnumerator(Of A)
Implements System.Collections.Generic.IEnumerable(Of A).GetEnumerator
        Return Me
    End Function

    Public ReadOnly Property Current As A Implements _
       System.Collections.Generic.IEnumerator(Of A).Current
        Get
            Try
                If _vec(indice) Is Nothing Then
                    Throw New InvalidOperationException("No hay elementos")
                End If
                Return _vec(indice)
            Catch ex As IndexOutOfRangeException
                Throw New InvalidOperationException( _
                    "No hay más elementos almacenados")
            End Try
        End Get
    End Property

    Public Function GetEnumerator1() As System.Collections.IEnumerator _
       Implements System.Collections.IEnumerable.GetEnumerator
        Throw New NotImplementedException()
    End Function

    Public ReadOnly Property Current1 As Object _
       Implements System.Collections.IEnumerator.Current
        Get
            Throw New NotImplementedException()
        End Get
    End Property

    Public Function MoveNext() As Boolean _
       Implements System.Collections.IEnumerator.MoveNext
        indice += 1
        Return indice < _vec.Length
    End Function

    Public Sub Reset() Implements System.Collections.IEnumerator.Reset
        indice = -1
```

```
    End Sub

#Region "IDisposable Support"
    Private disposedValue As Boolean ' To detect redundant calls

    ' IDisposable
    Protected Overridable Sub Dispose(disposing As Boolean)
        If Not Me.disposedValue Then
            If disposing Then
                ' TODO: dispose managed state (managed objects).
            End If
 ' TODO: free unmanaged resources (unmanaged objects) and override Finalize() below.
 ' TODO: set large fields to null.
        End If
        Me.disposedValue = True
    End Sub

    ' TODO: override Finalize() only if Dispose(ByVal disposing As Boolean) above
    ' has code to free unmanaged resources.
    ' Protected Overrides Sub Finalize()
    '  Do not change this code.  Put cleanup code in
    ' Dispose(ByVal disposing As Boolean) above.
    '    Dispose(False)
    '    MyBase.Finalize()
    'End Sub

    ' This code added by Visual Basic to correctly implement the disposable pattern.
    Public Sub Dispose() Implements IDisposable.Dispose
        ' Do not change this code.  Put cleanup code in
        ' Dispose(ByVal disposing As Boolean) above.
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
#End Region
End Class
```

Por otro lado, la contravarianza es el camino inverso para los parámetros de los tipos genéricos. Para que un parámetro sea contravariante debe ser el mismo tipo o un supertipo de éste. Análogamente, se pueden definir interfaces y delegados contravariantes, pero en este caso el parámetro deberá se especificado con la palabra clave `in`.

Ejemplo

C#

```
namespace contravarianza
{
    class ComparadorIgualdad : IEqualityComparer<B>
    {
        private int var;

        public ComparadorIgualdad(int var)
        {
            this.var = var;
        }
```

```
        public bool Equals(B x, B y)
        {
            return x.Equals(y);
        }

        public int GetHashCode(B obj)
        {
            return obj.GetHashCode();
        }

        public override bool Equals(object obj)
        {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (GetType() != obj.GetType())
                return false;
            ComparadorIgualdad otro = (ComparadorIgualdad)obj;
            if (var != otro.var) return false;
            return true;
        }

        public override int GetHashCode()
        {
            return var;
        }
    }
}

namespace contravarianza
{
    class Program
    {
        static void Main(string[] args)
        {
            // Las siguientes declaraciones son erróneas
            // porque los parámetros no tienen como
            // supertipo a B
            //IEqualityComparer<A> compA = new ComparadorIgualdad(1);
            //IEqualityComparer<C> compC = new ComparadorIgualdad(3);

            // Declaraciones correctas
            IEqualityComparer<B> compB = new ComparadorIgualdad(2);
            IEqualityComparer<D> compD = new ComparadorIgualdad(4);

            ImprimeHashCode(compB, compD);
            Console.WriteLine("------------------------------");
            // Asignaciones válidas por contravarianza
            IEqualityComparer<B> compB2 = compB;
            IEqualityComparer<D> compD2 = compB;

            ImprimeHashCode(compB2, compD2);
```

```
            Console.ReadKey();
        }

        public static void ImprimeHashCode(IEqualityComparer<D> obj,
              IEqualityComparer<D> obj2)
        {
            // Obtiene el código hash del objeto del
            // tipo ComparadorIgualdad
            D d1 = new D(5);
            D d2 = new D(5);
            Console.WriteLine("HashCode de d1 para obj: " + obj.GetHashCode(d1));
            Console.WriteLine("HashCode de d2 para obj2: " + obj2.GetHashCode(d2));
            if (obj.Equals(d1,d2))
                Console.WriteLine("Son Iguales");
            else
            {
                Console.WriteLine("Son diferentes");
            }
        }
    }
}
```

VB

```
Public Class ComparadorIgualdad
    Implements IEqualityComparer(Of B)

    Private _var As Integer

    Public Sub New(_var As Integer)
        Me._var = _var
    End Sub

    Public Overrides Function Equals(obj As Object) As Boolean
        If Me Is obj Then
            Return True
        End If

        If obj Is Nothing Then
            Return False
        End If
        If Me.GetType() <> obj.GetType() Then
            Return False
        End If

        Dim otro As ComparadorIgualdad = CType(obj, ComparadorIgualdad)

        If _var <> otro._var Then
            Return False
        End If
        Return True
    End Function

    Public Overloads Overrides Function GetHashCode() As Integer
        Return _var
    End Function
```

```
    Public Overloads Function Equals(x As B, y As B) As Boolean _
       Implements System.Collections.Generic.IEqualityComparer(Of B).Equals
        Return x.Equals(y)
    End Function

    Public Overloads Function GetHashCode(obj As B) As Integer _
       Implements System.Collections.Generic.IEqualityComparer(Of B).GetHashCode
        Return obj.GetHashCode()
    End Function
End Class

Module Module1

    Sub Main()
        ' Las siguientes declaraciones son erróneas
        ' porque los parámetros no tienen como
        ' supertipo a B
        'Dim compA As IEqualityComparer(Of A) = New ComparadorIgualdad(1)
        'Dim compC As IEqualityComparer(Of C) = new ComparadorIgualdad(3)

        ' Declaraciones correctas
        Dim compB As IEqualityComparer(Of B) = New ComparadorIgualdad(2)
        Dim compD As IEqualityComparer(Of D) = New ComparadorIgualdad(4)

        ImprimeHashCode(compB, compD)
        Console.WriteLine("-----------------------------")
        ' Asignaciones válidas por contravarianza
        Dim compB2 As IEqualityComparer(Of B) = compB
        Dim compD2 As IEqualityComparer(Of D) = compB

        ImprimeHashCode(compB2, compD2)

        Console.ReadKey()
    End Sub

    Public Sub ImprimeHashCode(obj As IEqualityComparer(Of D), _
       obj2 As IEqualityComparer(Of D))
        ' Obtiene el código hash del objeto del
        ' tipo ComparadorIgualdad
        Dim d1 As New D(5)
        Dim d2 As New D(5)
        Console.WriteLine("HashCode de d1 para obj: {0}", obj.GetHashCode(d1))
        Console.WriteLine("HashCode de d2 para obj2: {0}", obj2.GetHashCode(d2))
        If obj.Equals(d1, d2) Then
            Console.WriteLine("Son Iguales")
        Else
            Console.WriteLine("Son diferentes")
        End If
    End Sub
End Module
```