Capítulo 6
Tipos genéricos

1. Introducción

Los tipos genéricos permiten combinar la reutilización del código y la seguridad del tipo. Los tipos genéricos se usan más habitualmente para las colecciones. El Framework .NET expone estas colecciones en el espacio de nombres `System.Collections.Generic`, como el tipo `List<T>` o `Dictionary<TKey, TValue>`. Como es lógico, es posible crear sus propios tipos genéricos para crear una solución adaptada.

La ventaja de los tipos genéricos respecto a las colecciones clásicas como el tipo `ArrayList` es que el tipo de los objetos se conserva, mientras que una colección no genérica almacena los datos, haciendo una conversión al tipo `Object`. La recuperación de un elemento de una colección clásica obliga a hacer la conversión inversa, para hacer corresponder con el tipo esperado, mientras que los tipos genéricos devuelven un objeto ya tipado. A pesar de que la complejidad de codificación es ligeramente superior, los tipos genéricos además aportan mucha más rapidez, sobre todo cuando los elementos de la lista son tipos por valor.

2. La creación de tipos genéricos

Un tipo genérico se define en la declaración de la clase, indicando el parámetro `T`. Este parámetro especifica que el tipo lo elegirá el consumidor de la clase. Puede ser un tipo por valor o por referencia.

Cree una nueva clase `ReportChangeList<T>` en la carpeta **Library** del proyecto:

```
public class ReportChangeList<T>
{
}
```

El parámetro `T` acepta un tipo, que se especificará durante la instanciación:

```
ReportChangeList<int> Ex1 = new ReportChangeList<int>();
ReportChangeList<Object> Ex2 = new ReportChangeList<Object>();
```

Un tipo genérico también puede hacer referencia a varias clases:

```
public class ClaseGenerica<T, U>
```

Los tipos genéricos se pueden sobrecargar, siempre que el número de sus parámetros de tipo no sea idéntico:

```
public class ClaseGenerica<T>
public class ClaseGenerica<T, U>
```

Si dos tipos genéricos tienen el mismo nombre y el mismo número de parámetros de tipo, el compilador devolverá un error:

```
public class ClaseGenerica<T>
public class ClaseGenerica<U>          // No autorizado
```

La clase puede contener miembros que van a utilizar el tipo especificado en la instanciación, gracias al parámetro `T`. Añada el miembro `children` de tipo `List<T>` a la clase:

```
protected List<T> children;
```

Como `List<T>` es un tipo por referencia, se debe instanciar en el constructor para que no sea `null`.

Añada el constructor que instancia la variable `children`:

```
public ReportChangeList()
{
    this.children = new List<T>();
}
```

Durante la instanciación, la clase `ReportChangeList` instanciará una lista genérica con el tipo que se haya especificado.

El objetivo principal de crear un tipo genérico que contenga una lista genérica es poder limitar las funcionalidades expuestas o añadirlas. La clase `ReportChangeList` contendrá las clases hijas en su lista `children` y deberá notificar si el objeto contiene un cambio. Añada la interfaz `IReportChildrenChange` en la declaración de la clase:

```
public class ReportChangeList<T>: IReportChildrenChange
```

Como la clase hereda de la interfaz `IReportChildrenChange`, hay que añadir los miembros definidos en esta interfaz:

```
protected bool hasChanged;

public bool HasChanged
{
    get
    {
        bool result = false;
        foreach (IReportChange child in this.children)
            if (child.HasChanged)
            {
                result = true;
                break;
            }
        return this.hasChanged || result;
    }
    set
    {
        if (this.HasChanged != value)
        {
            this.hasChanged = value;
            if (this.Changed != null)
                this.Changed(this, new EventArgs());
        }
```

```
        if (!value)
        {
            foreach (IReportChange child in this.children)
                child.HasChanged = value;
        }
    }
}

public event EventHandler Changed;

public void ChildChanged(object sender, EventArgs e)
{
    if (this.Changed != null)
        this.Changed(sender, e);
}
```

En el código anterior, los descriptores de acceso de la propiedad `HasChanged` realizan un bucle sobre los elementos de la lista `children` para hacer la actualización de los objetos para el descriptor de acceso `set` o determinar si el objeto, en sí mismo o alguno de los elementos de la lista, se ha modificado por el descriptor de acceso `get`.

Para determinar si el objeto se ha modificado el principio es que si al menos se ha modificado un elemento de la lista o el objeto en sí mismo el objeto entero se considera como modificado. La actualización de la propiedad `HasChanged` de los elementos de la lista solo se lleva a cabo si el objeto recibe el valor `false` en la propiedad `HasChanged`, ya que el objeto se puede modificar pero no sus hijos mientras que, si los hijos se modifican, como consecuencia el objeto padre también se modifica, ya que los hijos forman parte del objeto.

3. Las restricciones de tipo

Como hemos visto en los descriptores de acceso, la interfaz `IReportChange` se usa para acceder a la propiedad `HasChanged` de los elementos de la lista. Esto significa que los elementos añadidos deben implementar obligatoriamente la interfaz. Para limitar los tipos que pueden reemplazar al parámetro genérico `T`, se pueden aplicar restricciones.

A continuación se muestran las restricciones existentes:

Restricción	Descripción
`where T:` *clase*	Restricción sobre la clase base del tipo.
`where T:` *interfaz*	Restricción sobre la interfaz que implementa el tipo.
`where T: class`	El tipo debe ser un tipo por referencia.
`where T: struct`	El tipo debe ser una estructura.
`where T: new()`	El tipo debe tener un constructor sin parámetro.
`where U: T`	El tipo representado por `U` debe ser idéntico al tipo `T`.

Añada una restricción a la interfaz `IReportChange` sobre la clase genérica `ReportChangeList`:

```
public class ReportChangeList<T>: IReportChildrenChange where T:
IReportChange
```

Las restricciones se aplican a todos los parámetros de tipo definidos, ya sea en los métodos o en la definición de tipo.

Las restricciones de tipo también se pueden definir a nivel de los métodos:

```
public void MiMetodo<T>() where T: class
{ }
```

4. Las interfaces genéricas

Como para las listas, sería interesante poder hacer un bucle `foreach` sobre los elementos de nuestro tipo genérico. Para esto es suficiente con implementar la interfaz genérica `IEnumerable<T>`:

```
public class ReportChangeList<T>: IReportChildrenChange,
IEnumerable<T> where T: IReportChange
```

Esta interfaz genérica se comporta como una interfaz clásica, contiene las declaraciones de los miembros necesarios para su implementación y se puede utilizar la clase genérica como referencia.

Añada los miembros `GetEnumerator()` e `IEnumerable.GetEnumerator()` siguientes, para implementar la interfaz en la clase genérica `ReportChangeList`:

```
public IEnumerator<T> GetEnumerator()
{
    for (int i = 0; i < this.children.Count; i++)
    {
        yield return this.children[i];
    }
}
IEnumerator IEnumerable.GetEnumerator()
{
    return GetEnumerator();
}
```

La implementación de un enumerador consiste en hacer un bucle sobre los elementos de la lista y devolver cada uno de ellos a la llamada. La palabra reservada `yield return` permite hacer un retorno a la llamada con el valor a devolver en función del valor anterior devuelto. El estado del método se mantiene de tal manera que puede continuar su ejecución hasta la siguiente llamada. El tiempo de vida de este estado está relacionado con el enumerador, el estado del método se elimina cuando la enumeración termina.

4.1 La varianza en las interfaces genéricas

La covarianza y la contravarianza son conceptos utilizados desde la versión 2.0 del Framework .NET. La versión 4 del Framework .NET añade la posibilidad de aplicar esos principios a las interfaces genéricas.

4.1.1 La covarianza

Una interfaz covariante permite a sus miembros devolver tipos más variados que aquellos que se han especificado. La interfaz `IEnumerable<T>`, implementada en la clase `ReportChangeList`, forma parte de las interfaces covariantes. Esto permite reutilizar los métodos que funcionan con las colecciones base genéricas para las colecciones derivadas.

El siguiente ejemplo ilustra la covarianza:

```
public class Clase1
{
    public string Prop1 { get; set; }
    public string Prop2 { get; set; }
    public virtual object Valores()
    {
        return $"{Prop1} {Prop2}";
    }
}
public class Clase2: Clase1

{
    public override string Valores ()
    {
        return $"{Prop1} {Prop2}";
    }
}

static class Program
{
    public static void Write(IEnumerable<Clase1> Elementos1)
    {
        foreach (Clase1 elem in Elementos1)
       {
            Console.WriteLine(elem.Valores());
        }
    }

    [STAThread]
    static void Main()
    {
        IEnumerable<Clase2> Elementos2 = new List<Clase2>();
        Write(Elementos2);
    }
}
```

El método `Write` acepta una colección de tipo `IEnumerable<Clase1>` como parámetro. La covarianza permite llamar al método con una colección de tipo `IEnumerable<Clase2>`, ya que el tipo `Clase2` hereda del tipo `Clase1`.