Sobrecarga de operadores y genéricos

Índice

1 Introducción

La sobrecarga de operadores en C# se realiza mediante métodos estáticos, basicamente empleando la sintaxis operator<operador>, y como argumentos el tipo a la izquierda y el tipo a la derecha (si es un operador binario), o el tipo sobre el que actúa (si es unario):

Sintaxis (operador binario)

```
public static <tipo_retorno> operator<operador>(<tipo_op1> op1, <tipo_op2> op2)
{
      // más cosas...
}
```

Ejemplo

```
class Integer {
    public Integer(int x)
    {
          this.Value = x;
    }

    public int Value {
          get; set;
    }

    // Se invoca con: i1 + i2, siendo i1 e i2 de la clase Integer
    public static Integer operator+(Integer a, Integer b)
    {
         return new Integer( a.Value + b.Value );
    }

    // más cosas...
}
```

Sintaxis (operador unario)

```
public static <tipo_retorno> operator<operador>(<tipo_op1> op1)
{
      // más cosas...
}
```

Ejemplo

```
class Integer {
    public Integer(int x)
    {
          this.Value = x;
    }

    public int Value {
          get; set;
    }

    // Se invoca con i1++ o ++i1, siendo i1 de la clase Integer
    public static Integer operator++(Integer a)
    {
         return new Integer( a.Value + b.Value );
    }
```

```
    // más cosas...
}
```

En el caso de los operadores unarios ++ o –, hay que recordar que el operador postfijo se comporta como si primero devolviera el valor actual del objeto, y solo después hiciera el incremento o decremento. Este comportamiento se obtiene en la práctica haciendo una copia antes de realizar el incremento o decremento, efectuar la operación, y después devolver la copia realizada. De esto se encarga el *runtime* de C# automáticamente, de forma que una sola sobrecarga es suficiente tanto para las versiones prefijas como postfijas.

En realidad, aparte de los operadores unarios ++ y –, no puede sobrecargarse ningún otro operador que implique una asignación. Así, operadores como =, +=, -= o *= no es posible sobrecargarlos, aunque sí es posible sobrecargar +, -, *… teniendo en cuenta que una instrucción como `x += 5` se descompone en realidad en `x = x + 5`, esto no supone mayor problema.

Un tipo de operador unario especial es el de conversión explícita o *cast*. Es el que se invoca si se ejecutara, por ejemplo, `(Integer) 5`.

Sintaxis

```
public static explicit operator<nombre_del_tipo>(<tipo_op1> op1)
{
      // más cosas...
}
```

Ejemplo

```
class Integer {
    public Integer(int x)
    {
          this.Value = x;
    }

    public int Value {
          get; set;
    }

    // Se invoca con (Integer) 5.
    public static explicit operator Integer(int x)
    {
         return new Integer( x );
    }

    // Se invoca con (int) i1, siendo i1 de la clase Integer.
    public static explicit operator int(Integer x)
    {
         return x.Value;
    }

    // más cosas...
}
```

2 Comparaciones

En principio, para poder tener operadores como == funcionando, solo es necesario sobrecargar el mismo operador == (véase la siguiente sección). Sin embargo, cuando empezamos con los operadores de igualdad, debemos enseguida pensar en dos métodos: **Object**.*Equals(object o)* y

Object.*GetHashCode()*. Estos dos métodos son los que se usarían cuando se introduce un objeto de una de nuestras clases en, por ejemplo, una colección como **Map**. Además, es interesante también implementar la interfaz **IComparable** (consiste en solo un método, *CompareTo(object o)*), e **IComparable<Clase>**, siendo Clase nuestra propia clase.

2.1 El método Equals()

El método *Equals(object o)* comprueba si los atributos de un objeto son iguales a los de otro. Este método ya existe en la clase base **Object**, así que debe ser reescrito (*override*).

```
class Integer  {
    public override bool Equals(object obj)
    {
        Integer op2 = obj as Integer;

        if ( op2 == null ) {
            return false;
        }

        return this.Value == op2.Value;
    }
    // ...
}
```

En el caso de un objeto que no pertenezca a nuestra clase, el resultado debe ser falso. Solo en el caso de que el objeto pertenezca a nuestra clase, realizamos algún tipo de comparación.

2.2 El método GetHashCode()

El método *GetHashCode()* devuelve un valor numérico. El valor devuelto debe cumplir una serie de propiedades, como la de ser el mismo valor para un mismo conjunto de atributos, de manera que si dos objetos tienen los mismos valores en sus atributos, su código hash debe ser el mismo. Este método ya existe en la clase base **Object**, así que debe ser reescrito (*override*).

```
class Integer  {
    public override int GetHashCode()
    {
        return this.Value.GetHashCode();
    }
    // ...
}
```

En el caso concreto de la clse **Integer**, devolver el código hash es tan sencillo como devolver el código hash del único atributo, pero… ¿qué hacer si la clase contiene más de un atributo?. En el siguiente ejemplo, se muestra la clase **Point**.

```
class Point  {
    public override bool Equals(object obj)
    {
        Point op2 = obj as Point;

        if ( op2 == null ) {
            return false;
        }

        return this.X == op2.X && this.Y == op2.Y;
```

```
    }

    public override int GetHashCode()
    {
        return ( 11 * this.X.GetHashCode() ) + this.Y.GetHashCode();
    }

    public int X {
        get; set;
    }

    public int Y {
        get; set;
    }

    // ...
}
```

Los códigos *hash* en estos casos se calculan utilizando números primos (como 11), para multiplicar cada uno de los atributos, y finalmente sumar todos ellos.

2.3 La interfaz IComparable

La interfaz **IComparable** existe en dos versiones, la que actúa sobre un segundo argumento **Object**, y la que es una template que actúa sobre un segundo argumento de la misma clase que la implementa.

Esta interfaz se emplea, por ejemplo, cuando tenemos un vector *v* de objetos de nuestra clase, y se lllama a **Array**.*Sort(v)*.

Se trata de un método en el que si **this** es menor que el parámetro, devuelve -1, 0 si son iguales, y 1 en el caso en el que el parámetro es mayor que **this**.

```
class Integer: IComparable, IComparable<Integer>  {
    public int CompareTo(object obj)
    {
        int toret = 1;
        Integer op2 = obj as Integer;

        if ( op2 != null ) {
            if ( this.Value < op2.Value ) {
                toret = -1;
            }
            else
            if ( this.Value == op2.Value ) {
                toret = 0;
            }
            else
            if ( this.Value > op2.Value) {
                toret = 1;
            }
         }

         return toret;
    }

    public int CompareTo(Integer x)
    {
```

```
        return this.Value.CompareTo( x.Value );
    }

    // ...
}
```

Dado que se muestran dos métodos que al fin y al cabo hacen lo mismo, se muestran dos posibles implementaciones. La primera compara ambos objetos (**this** y *obj*), devolviendo -1, 0 y 1 en el caso en el que **this** sea menor que *obj*, sean iguales, o **this** mayor que *obj*, respectivamente. El segundo se aprovecha de que los tipos primtivos ya tienen su propia versión de *CompareTo()*, por lo que devuelve directamente el resultado de llamar a *CompareTo()* del valor entero almacenado en la propiedad *Value*.

2.4 Operadores relacionales

Los operadores relacionales son los siguientes: <, > != ==, >=, y <=. En realidad, la sobrecarga de estos operadores no difiere en nada del resto de operadores binarios, pero es mejor sobrecargarlos en función de *Equals(object o)* y *compareTo(object o)*. Es importante tener en cuenta que cada método de ser sobrecargado por separado, es decir, sobrecargar == no implica que != esté sobrecargado, o vicecersa, o que al sobrecargar <, el operador > ya esté sobrecargado. De hecho, sobrecargar == sin sobrecargar !=, por ejemplo, provoca un error.

```
class Integer: IComparable, IComparable<Integer>  {
    public static bool operator==(Integer a, Integer b)
    {
        if ( ( (object) a ) == null ) {
             return false;
        }

        return a.Equals( b );
    }

    public static bool operator!=(Integer a, Integer b)
    {
        return !( a == b );
    }

    // ...
}
```

El *cast* de *a* a **object** antes de comprobar si a es null puede parecer caprichoso, pero está ahí por un motivo: el hecho es que manteniendo el tipo Integer de a al comparalo con null, estaríamos llamando recursivamente al operador ==, hasta agotar la pila.

El siguiente código se refiere a la sobrecarga de < y >. Los operadores >= y <= pueden sobrecargarse en función de los ya creados.

```
class Integer: IComparable, IComparable<Integer>  {
      public static bool operator>(Integer a, Integer b)
      {
            if ( ( (object) a ) == null ) {
                  return false;
            }

            return a.CompareTo( b ) > 0;
      }

      public static bool operator<(Integer a, Integer b)
```

```
    {
        return !( a == b || a > b );
    }

    // ...
}
```

3 Genéricos

Las clases genéricas son aquellas que permiten ser aplicadas a distintos tipos. Por ejemplo, las colecciones (por ejemplo, una lista), en C# en **System.Collections.Generic** son genéricas, es decir, pueden aplicarse a cualesquiera tipos manteniendo una comprobación de errores en tiempo de compilación.

La comprobación de errores es importante, ya que las mismas colecciones tienen una versión no genérica en **System.Collections**, que actúa sobre **object**. La única ventaja es que se pueden mezclar objetos de distintas clases dentro de una colección, pero se pierde totalmente la posibilidad de comprobación de errores, ya que el compilador solo sabe que la colección almacena instancias de **object**, y recordemos que en C# todas las clases derivan de **Object**.

Para crear una clase genérica simplemente se sigue el nombre de la clase con <T>, siendo T el tipo sobre el que actuará la clase. Puede haber varios tipos, en cuyo caso se especificarán separados por comas como T, U…

```
class Vector<T> {
	public Vector(int capacity = 1)
	{
		this.v = new T[ capacity ];
		this.Size = 0;
	}

	public int Capacity {
		get {
			return this.v.Length;
		}
	}

	public int Size {
		get; private set;
	}

	public void Add(T x)
	{
		this.Resize();
		this.v[ this.Size ] = x;
		++this.Size;
	}

	public T Get(int i)
	{
		ChkRange( i );
		return this.v[ i ];
	}

	public void Set(int i, T x)
	{
		ChkRange( i );
		this.v[ i ] = x;
	}

	private void Resize()
	{
		if ( this.Size >= this.Capacity ) {
			T[] v2 = new T[ this.Capacity * 2 ];
			Array.Copy( this.v, v2, this.v.Length );
			this.v = v2;
		}
```

```
            return;
      }

      private T[] v;
}
```

La sobrecarga de operadores es muy interesante en cuanto a estas clases genéricas, pues es muy común que se pretenda que actúen de forma indistinguible a las clases que provee la librería estándar.

```
class Vector<T> {
      // v[i], v is Vector
      public T this[int x] {
            get {
                  return this.Get( x );
            }
            set {
                  this.Set( x, value );
            }
      }

      // v1 + v2, v1 is Vector && v2 is Vector
      public static Vector<T> operator+(Vector<T> v1, Vector<T> v2)
      {
            var toret = new Vector<T>( v1.Size + v2.Size );

            for(int i = 0; i < v1.Size; ++i) {
                  toret.Add( v1[ i ] );
            }

            for(int i = 0; i < v2.Size; ++i) {
                  toret.Add( v2[ i ] );
            }

            return toret;
      }

      // (List<T>) v, v is Vector<T>
      public static explicit operator List<T>(Vector<T> v)
      {
            var toret = new List<T>();

            for(int i = 0; i < v.Size; ++i) {
                  toret.Add( v[ i ] );
            }

            return toret;
      }

      // ...
}
```