11 INTERFACES

Como se explica en el Capítulo de Herencia, uno de los beneficios de la POO es la factorización de código. Mediante las jerarquías de tipo que se pueden formular con la herencia, una clase base expresa la funcionalidad común que puede ser heredada, ampliada o adaptada por las clases derivadas que heredan de ella.

El concepto de **interface**[1] que se estudia en este capítulo permite definir un tipo en su expresión más abstracta indicando solo las funcionalidades que debe tener, las cuales luego deberán ser implementada por aquellas clases que digan que implementan esa interface.

Observe el código del Listado 11-1 y el Listado 11-2, el primero implementa la ordenación de un array de enteros, el segundo intenta hacer lo mismo para ordenar un array de fechas.

```
static void Ordenar(int[] items) {
  for (int k = 0; k < items.Length - 1; k++)
    for (int j = k + 1; j < items.Length; j++)
      if (items[j] < items[k]) {
        int temp = items[j];
        items[j] = items[k];
        items[k] = temp;
      }
}
```

Listado 11-1 Ordenar un array de enteros

```
static void Ordenar(Fecha[] items) {
  for (int k = 0; k < items.Length - 1; k++)
    for (int j = k + 1; j < items.Length; j++)
      if (items[j] < items[k]) {
        Fecha temp = items[j];
        items[j] = items[k];
        items[k] = temp;
      }
}
```

Listado 11-2 Ordenar un array de fechas

[1] La traducción al español del término *interface* sería realmente *interfaz*, pero hemos decidido utilizar el término inglés para mantener mayor coherencia con la palabra clave interface utilizada en C#. Esto no tiene nada que ver con el significado del término en español *interfase* y que a veces es utilizado erróneamente por algunos.

Pero el código del Listado 11.2 es erróneo porque la comparación `items[j] < items[k]` no está definida para el tipo `Fecha`. Sin embargo, note que el código de la funcionalidad `Ordenar` es similar al de ordenar un array de `int`, y la diferencia solo consiste en la forma de comparar dos elementos del array. Para el tipo `int` existe la operación de comparación `<`, pero para el tipo `Fecha` no está definida esta operación. Sería ingenuo pretender que todos los tipos tengan definida una tal operación de comparación cuando incluso para muchos tipos esto no aportaría ninguna utilidad, ¿qué provecho tendría, por ejemplo, pretender ordenar un array de objetos de tipo `Stopwatch`?

¿Cómo un tipo podría indicar que un objeto de dicho tipo es susceptible de compararse con otro del mismo tipo y por tanto poder aprovechar la existencia de una funcionalidad como la de `Ordenar`? La solución a esto se expresará con la interface `IComparable<T>`.

11.1 Interface IComparable<T>

El Listado 11-3 nos muestra el código de la interface `IComparable<T>`. Esta es una interface ya ofrecida por .NET en el espacio de nombres `System`. Una interface se define con una sintaxis parecida a la de definir una clase, pero utilizando la palabra `interface` en lugar de `class`. A diferencia de la definición de una clase, en una interface solo se pueden especificar las signaturas de métodos y propiedades, pero no se pueden especificar variables. Note que en este ejemplo esta interface solo contiene el método `CompareTo(T other)`

```
namespace System
{
  public interface IComparable<in T>
  {
    int CompareTo(T other);
  }
}
```

Listado 11-3 Interface IComparable<T>

Una clase que define un tipo que quiera gozar de la funcionalidad especificada por la interface debe entonces indicar que **implementa** dicha interface e implementar los métodos y propiedades que se especifican en ella. Así, el código del Listado 11-4 indica que la clase `Fecha` implementa la interface `IComparable<Fecha>`. Esto implica que si `f1` y `f2` son dos objetos de tipo `Fecha`, entonces estos se pueden comparar haciendo `f1.CompareTo(f2)`.

```
class Fecha: IComparable<Fecha>
{
  public int D { get; private set; }
  public int M { get; private set; }
  public int A { get; private set; }

  //... Otros Métodos de Fecha

  public int CompareTo(Fecha f)
  {
    if (A < f.A) return -1;
    else if (A > f.A) return 1;
    if (M < f.M) return -1;
```

```
    else if (M > f.M) return 1;
    if (D < f.D) return -1;
    else if (D > f.D) return 1;
    else return 0;
  }
}
```

Listado 11-4 Fecha implementa IComparable<Fecha>

El convenio adoptado en la implementación del método CompareTo es

si `f1.CompareTo(f2)` es `<0` entonces `f1` es menor que `f2`

si `f1.CompareTo(f2)` es `>0` entonces `f1` es mayor que `f2`

si `f1.CompareTo(f2)` es `0` entonces `f1` es igual a `f2`

De este modo, el método `Ordenar` podría implementarse como se muestra en el Listado 11.5. Al especificar

```
static void Ordenar<T>(T[] items) where T: IComparable<T>
```

se indica como restricción que el tipo `T`, que se usa para definir el tipo de los elementos del array que se quiere ordenar, tiene a su vez que implementar la interface `IComparable<T>` y por tanto implementar el método `CompareTo`. Por consiguiente, la comparación de la línea 4 del Listado 11-5 es correcta y aplicará la implementación que `T` haya hecho del método `CompareTo`.

```
static void Ordenar<T>(T[] items) where T: IComparable<T> {
  for (int k = 0; k < items.Length - 1; k++)
  for (int j = k + 1; j < items.Length; j++)
    if (items[j].CompareTo(items[k])<0) {
      T temp = items[j];
      items[j] = items[k];
      items[k] = temp;
    }
}
```

Listado 11-5 Implementación del método Ordenar basándose en que los elementos del array son IComparables

Si según el código del Listado 11-4 `Fecha` es `IComparable<Fecha>`, entonces si se tiene

```
Fecha[] efemerides = {new Fecha(1,1,2010), new Fecha(31, 12, 2010),
      new Fecha(1,5,2011), new Fecha(25,12,2010), new Fecha(31,1,2009)};
```

entonces se puede ordenar dicho array invocando a `Ordenar(efemerides)`.

Este ejemplo nos ilustra un patrón importante de factorización y reutilización que se logra mediante las interfaces. Se escribe un único código de `Ordenar` basándose en que los elementos del array son comparables; quien quiera aprovecharse de la existencia de este método no tiene que preocuparse de cómo éste está implementado, sino que solo tiene que implementar la interface `IComparable<T>`.

Decir que somos de un tipo interface es como asociarse a un club, tenemos la obligación de pagar la cuota de inscripción (implementar los métodos de la interface) pero tenemos

los beneficios de usufructuar los servicios del club (al poder usar la funcionalidad de un código que se base en dicha interface). Si soy del club de los *comparables* puedo usar el servicio de *ordenación*.

Si el tipo `Cuenta` del Listado 11-6 implementa `IComparable<Cuenta>`, entonces se puede ordenar un array de cuentas usando el método `Ordenar` implementado para cualquier `T` que sea `IComparable<T>`.

```
Public class Cuenta: IComparable<Cuenta>
{
  public float { get; protected set; }
  public string IComparer { get; private set; }

  //...Métodos de Cuenta...

  public int CompareTo(Cuenta c)
  {
    if (Saldo < c.Saldo) return -1;
    else if (Saldo > c.Saldo) return 1;
    else return 0;
  }
}
```

Listado 11-6 Implementación de Cuenta como IComparable<Cuenta>

Si se tiene

```
Cuenta[] cuentas = {new Cuenta("Miguel", 1000), new Cuenta("Juan", 500),
                    new Cuenta("Luis", 2000), new Cuenta("Ana", 500)};
```

se puede entonces ordenar dicho array haciendo `Ordenar(cuentas)`

El tipo integrado `string` implementa la interface `IComparable<string>` de modo que el siguiente código es correcto

```
string[] colores = {"rojo", "azul", "blanco", "negro", "amarillo"};
Ordenar(colores);
```

11.2 Método System.Array.Sort usa IComparable

.NET ofrece en la clase `System.Array` un método de ordenación `Sort` basado en que el tipo `T` de los elementos del array implementen `IComparable<T>`. El código del Listado 11-7 es también correcto

```
Fecha[] efemerides = {new Fecha(1,1,2010), new Fecha(31, 12, 2010),
  new Fecha(1,5,2011), new Fecha(25,12,2010),
  new Fecha(31,1,2009)};
System.Array.Sort(efemerides);
Cuenta[] cuentas = {new Cuenta("Miguel", 1000), new Cuenta("Juan", 500),
  new Cuenta("Luis", 2000), new Cuenta("Ana", 500)};
System.Array.Sort(cuentas);
string[] colores = {"rojo", "azul", "blanco", "negro", "amarillo"};
```

```
System.Array.Sort(colores);
```

Listado 11-7 Usando Array.Sort

11.3 Interface ICloneable

Otra interface de utilidad es `ICloneable`, que ofrece un método que hace un clon o duplicado de un objeto (Listado 11-8). Hacer un duplicado de un objeto es útil en escenarios en los que queremos realizar operaciones sobre un objeto que pudieran modificarlo, pero queremos mantener una copia del objeto en su estado original.

```
public interface ICloneable
{
  object Clone();
}
```

Listado 11-8 Interface ICloneable

La clase `Fecha` del Listado 11-9 implementa la interface `ICloneable`, note que una clase puede implementar más de una interface (del mismo modo que Ud. puede ser miembro de diferentes clubs y aprovecharse de los servicios de cada uno).

```
class Fecha: IComparable<Fecha>, ICloneable
{
  public int D { get; private set; }
  public int M { get; private set; }
  public int A { get; private set; }

  //...Otros métodos de Fecha....

  public Fecha(int d, int m, int a)
  {
    D = d; M = m; A = a;
  }

  //...public int CompareTo(Fecha f)

  public object Clone()
  {
      return new Fecha(D, M, A);
  }
}
```

Listado 11-9 Fecha es ICloneable e IComparable

El método `CloneArray` del Listado 11-10 recibe un array de elementos que implementen `ICloneable` y devuelve un array del mismo tamaño y tipo donde cada elemento es un clon o duplicado del array original.

```
static T[] CloneArray<T>(T[] items) where T : ICloneable
{
  T[] result = new T[items.Length];
  for (int k = 0; k < items.Length; k++)
    if (items[k] != null)
      result[k] = (T)(items[k].Clone());
  return result;
}
```

Listado 11-10 Clonación de un array

Note la operación de casting `result[k] = (T)(items[k].Clone);` es necesario hacer esto porque el método `Clone` de `ICloneable` devuelve un objeto de tipo `object` y queremos reafirmar que estamos devolviendo un array de elementos de tipo `T`.

.NET no ofrece una interface genérica `ICloneable<T>`, pero podemos definir una tal interface como

```
interface ICloneable<out T>
{
  T Clone();
}
```

La especificación `out` que acompaña al parámetro genérico T significa que dentro de la interface el parámetro solo puede aparecer como tipo de retorno de un método o propiedad o como parámetro genérico de otro tipo en el que a su vez sea también especificado.

De este modo, es mejor implementar `Fecha` como `ICloneable<Fecha>` tal y como se muestra en el Listado 11-11. Note que ahora la operación `Clone` aplicada a una entidad de tipo `Fecha` devuelve un objeto de tipo `Fecha`. Es decir, si `f` es de tipo `Fecha` es correcto hacer `Fecha copia = f.Clone()` sin necesidad de hacer una operación de casting. La implementación del método `CloneArray` quedaría mejor ahora como se muestra en el Listado 11-12; al tener que ser `T` un tipo que cumpla con `ICloneable<T>`, entonces `items[k].Clone()` devuelve un objeto de tipo `T` y por consiguiente el compilador no necesita que se coloque una operación de casting.

```
class Fecha: IComparable<Fecha>, ICloneable<Fecha>
{
  //...
  public Fecha Clone()
  {
      return new Fecha(D, M, A);
  }
}
```

Listado 11-11 Fecha como ICloneable<Fecha>

```
static T[] CloneArray<T>(T[] items) where T : ICloneable<T>
{
  T[] result = new T[items.Length];
  for (int k = 0; k < items.Length; k++)
    if (items[k] != null)
      result[k] = items[k].Clone(); //No es necesario el casting
  return result;
}
```

Listado 11-12 Nueva implementación de CloneArray

11.4 Implementación implícita y explícita de una interface

Si una clase puede implementar más de una interface, puede ocurrir que varias de las interfaces tengan un método con el mismo nombre. Considere una clase `A` que implementa las interfaces `I1` e `I2` (Listado 11-13).

```
interface I1                interface I2
{                           {
  void M();                   void M();
  void F();                   void H();
}                           }
```

Listado 11-13 Interfaces I1 e I2

Ambas interfaces tienen un método común `M`. Tenemos varias interpretaciones posibles según el contexto en que se esté usando un objeto de tipo `A`. Si se implementa `A` como se muestra en el Listado 11-14, entonces la implementación de `M` vale como implementación común para todas las situaciones.

```
class A : I1, I2
{
public void M()
  {
      Console.WriteLine("Soy M para A, I1 e I2");
  }

public void F()
  {
      Console.WriteLine("Soy F para A y para I1");
  }

public void H()
  {
      Console.WriteLine("Soy H para A y para I2");
  }
}
```

Listado 11-14 Implementación implícita de dos interfaces con un método común

Si hacemos

```
A a = new A(); I1 ai1 = a; I2 ai2 = a;
```

y luego hacemos

```
a.M(); ai1.M(); ai2.M();
```

estamos en los tres casos invocando al mismo método `M` implementado en `A`.

Esto es lo que se conoce como **implementación implícita**. Al no dar ninguna indicación adicional, se asume que la implementación de `M` en `A` vale por igual para los clientes de `A` de `I1` y de `I2`.

Sin embargo, la implementación de `A` podría ser como la del Listado 11-15.

```
class A : I1, I2
{
  public void M()
  {
       Console.WriteLine("Soy M para A y para I1");
  }
  void I2.M()
  {
       Console.WriteLine("Soy M para I2");
  }
  public void F()
  {
       Console.WriteLine("Soy F para A y para I1");
  }

  public void H()
  {
       Console.WriteLine("Soy H para A y para I2");
  }
}
```

Listado 11-15 Implementación explícita para dos interfaces con un método común

En este caso, al escribir `void I2.M()`se está indicando que ese es el método que quiere ejecutarse cuando un objeto de tipo `A` se use como de tipo `I2`. Esto es lo que se conoce como **implementación explícita** de la interface. La ejecución del código

```
A a = new A(); I1 ai1 = a; I2 ai2 = a;
a.M(); ai1.M(); ai2.M();
```

daría la salida que se muestra en la Figura 11-1 Salida de una implementación explícitaFigura 11-1

```
Soy M para A y para I1
Soy M para A y para I1
Soy M para I2
```

Figura 11-1 Salida de una implementación explícita

Como se muestra en el ejemplo del Listado 11-15, la implementación explícita de un método no puede ser pública. Es decir, un cliente que use un `a` de tipo `A` no puede invocar directamente a dicha implementación haciendo algo como `a.I2.M()`; tendría que indicarlo explícitamente haciendo `((I2)a).M()`

11.5 La interface IComparer<T>

La utilización de la interface `IComparable<T>` que se hace en la implementación del método `Ordenar` del Listado 11-5 permite que el código de `Ordenar` sea reutilizable por cualquiera que quiera ordenar un array de elementos de tipo `T` siempre que `T` sea un tipo que implemente la interface `IComparable<T>`. De este modo, se pudo ordenar lo mismo un array de fechas, que un array de strings, que un array de cuentas.

La limitación de este enfoque es que el criterio de comparación (y por tanto el criterio de ordenación) lo pone la implementación que el tipo `T` haga de `IComparable<T>`. Así, por ejemplo, para el tipo `Cuenta` (Listado 11-6) la comparación es en base al saldo de la cuenta: si se ordena un array de tipo `Cuenta`, los elementos se ordenarán según la propiedad `Saldo`. ¿Y si quisiéramos ordenar según el nombre del titular de la `Cuenta`? Es obvio que no es posible tener en el tipo `Cuenta` dos implementaciones diferentes del mismo método `CompareTo`. Sin embargo, el código del algoritmo de ordenación es similar; la idea es basar la ordenación no en que los elementos de la cuenta se sepan comparar entre sí, sino suministrarle por quien invoca a `Ordenar` un objeto "comparador" que sepa "comparar" dos cuentas.

La interface `IComparer<T>` (Listado 11-16) define un tipo que aporta un método que compara a dos objetos de tipo `T`. El convenio asumido para la implementación del método `Compare` es similar al del método `CompareTo`, es decir, si `a` y `b` son de tipo `T` y c es de tipo `IComparer<T>`, entonces

si `c.Compare(a,b)` es `<0` entonces `a` es menor que `b`
si `c.Compare(a,b)` es `>0` entonces `a` es mayor que `b`
si `c.Compare(a,b)` es `0` entonces `a` es igual a `b`

```
public interface IComparer<in T>
{
  int Compare(T x, T y);
}
```

Listado 11-16 Interface IComparer

Se podría sobrecargar entonces la implementación de un método como `Ordenar` de manera que reciba también un parámetro de tipo `IComparer` que será quien realice la comparación (Listado 11.16). Note que, a diferencia de la implementación de Ordenar del Listado 11.5, aquí no se exige que el tipo `T` de los elementos del array sea `IComparable<T>`, ya que para la comparación se usará el parámetro `IComparer<T>`.

```
static void Ordenar<T>(T[] items, IComparer<T> comp)
{
  for (int k = 0; k < items.Length - 1; k++)
    for (int j = k + 1; j < items.Length; j++)
      if (comp.Compare(items[j],items[k]) < 0)
      {
        T temp = items[j];
        items[j] = items[k];
        items[k] = temp;
      }
}
```

Listado 11-17 Método Ordenar usando un IComparer

De este modo, si tenemos

```
Cuenta[] cuentas = {new Cuenta("Miguel", 1000), new Cuenta("Juan", 500),
  new Cuenta("Luis", 2000), new Cuenta("Ana", 500)};
```

y queremos ordenar este array según el nombre del titular, se podría llamar al método `Ordenar` del Listado 11-17 si se le pasase también como parámetro un objeto de tipo `IComparer<Cuenta>`. La clase `TitularCuentaComparer` del Listado 11-18 es una implementación de un comparador de cuentas. Note que a su vez se está usando el método `CompareTo` de `string`, ya que `IComparer` es de tipo `string` y el tipo `string` es `IComparable`

```
class TitularCuentaComparer: IComparer<Cuenta>
{
  public int Compare(Cuenta c1, Cuenta c2)
  {
    //...
    if (c1 == null || c2 == null)
      throw new ArgumentException("Parámetros de Compare no pueden ser null");
    return c1.IComparer.CompareTo(c2.IComparer);
  }
}
```

Listado 11-18 Implementación de un IComparer<Cuenta> según el `IComparer`

De modo que para ordenar el array `cuentas` bastaría con invocar

`Ordenar(cuentas,` new `TitularCuentaComparer()).`

Este es un enfoque más reutilizable que el de que el tipo `T` tenga que ser `IComparable<T>`. Es frecuente encontrarse con un escenario en el que se tiene una colección (por ahora un array) de elementos de tipo `T` y se quiere ordenar esta colección según diferentes criterios del tipo `T`. No tiene por qué ser decisión del implementador del tipo `T` por cuál criterio se va a ordenar; eso mejor lo decide quién decide ordenar. Considere, por ejemplo, el fragmento de implementación del tipo `Email` del Listado 11-19 Una lista de emails se puede querer ver ordenada por el remitente, por el asunto, por la fecha de envío, por la fecha de recepción, por el tamaño del mensaje, etc., y esto por lo general no tiene que ser una decisión de quien define el tipo `Email` sino de la parte de la aplicación que trabaja con la lista de emails.

```
class Email
{
  public string Sender { get; set; }
  public string Subject { get; set; }
  public long Size { get; set; }
  public DateTime Received { get; set; }
  public DateTime Sended { get; set; }
  //...
```

```
}
```

Listado 11-19 Implementación de un tipo Email

Habría que definir un `IComparer` por tantos criterios como se quiere realizar la ordenación, como se muestra en el Listado 11-20

```
class EmailSenderComparer: IComparer<Email>
{
  public int Compare(Email e1, Email e2)
  {
    if (e1 == null || e2 == null)
      throw new ArgumentException("Argumentos a comparar no pueden ser nulos");
    return e1.Sender.CompareTo(e2.Sender);
  }
}

class EmailSubjectComparer : IComparer<Email>
{
  public int Compare(Email e1, Email e2)
  {
    if (e1 == null || e2 == null)
      throw new ArgumentException("Argumentos a comparar no pueden ser nulos");
    else
      return e1.Subject.CompareTo(e2.Subject);
  }
}

class EmailSizeComparer : IComparer<Email>
{
  public int Compare(Email e1, Email e2)
  {
    if (e1 == null || e2 == null)
      throw new ArgumentException("Argumentos a comparar no pueden ser nulos");
    return (e1.Size.CompareTo(e2.Size));
  }
}

class EmailReceivedComparer : IComparer<Email>
{
  public int Compare(Email e1, Email e2)
  {
    if (e1 == null || e2 == null)
      throw new ArgumentException("Argumentos a comparar no pueden ser nulos");
    else
      return e1.Received.CompareTo(e2.Received);
  }
}

class EmailSendedComparer : IComparer<Email>
{
  public int Compare(Email e1, Email e2)
  {
    if (e1 == null || e2 == null)
      throw new ArgumentException("Argumentos a comparar no pueden ser nulos");
```

```
    else
      return e1.Sended.CompareTo(e2.Sended);
  }
}
```

Listado 11-20 Distintos comparadores de Email

Para clases como las que implementen `IComparer` bastaría con tener una sola instancia, cuyo único rol sería realizar la función de comparación. Usted puede pensar, y con razón, que es un poco fastidioso tener que definir toda una clase por cada criterio por el que se le ocurra querer comparar. Aunque si bien es cierto que se logra con ello reusar con facilidad el código de ordenar, ¿no podríamos aspirar a algo mejor? En el Capítulo 12 sobre Programación Funcional se verá otra alternativa.

11.6 Iteradores

Como se expone en el Capítulo 7, un array es la forma más primaria de representar en memoria una colección de datos. Si `a` es un array de un tipo `T`, es decir `a` es del tipo `T[]`, entonces los elementos del array se pueden recorrer de dos formas:

1) `for (int k=0; k<a.Length; k++)`
   { ...hacer algo con `a[k]` ... }

2) `foreach (T x in a)`
   { ...hacer algo con x ... }

En el primer caso, el código del ciclo `for` es el responsable no solo de "procesar" los elementos del array, sino que también especifica cómo se recorren los elementos de este. En el segundo caso, el código del `foreach` se desentiende de cómo se recorren los elementos del array y solo se ocupa de procesar estos. Ambos casos corresponden a uno de los patrones más comunes en la programación: por una parte tenemos *una fuente de datos* (el array en este ejemplo) y por otra parte tenemos un código *consumidor de dicha fuente de datos.*

En este epígrafe se presentan dos importantes interfaces que son una abstracción de una fuente de datos y que nos independizan de la forma en que se representan y se producen los datos. Estas interfaces son `IEnumerator<T>` e `IEnumerable<T>` (aunque existen sus versiones no genéricas `IEnumerator` e `IEnumerable`).

11.6.1 Interfaces IEnumerator<T> e IEnumerator

La interface `IEnumerator<T>` (Listado 11-21) expresa un "motor" de recorrido (o producción) de datos.

Se le suele decir un motor de recorrido cuando los datos están físicamente representados en memoria en alguna estructura y la implementación de la interface lo que indica es cómo se recorren. Se le suele decir un motor de producción cuando los datos no necesariamente están físicamente representados en memoria, sino que se van produciendo por demanda según se van procesando. Pero de manera coloquial se utilizan indistintamente ambos términos.

Esta interface expresa de modo minimalista una fuente de datos. Si se tiene un objeto c de algún tipo que implemente `IEnumerator<T>`, entonces si `c.MoveNext()` devuelve true eso indica que la fuente de datos tiene un dato listo para ofrecer y que lo podemos obtener haciendo `c.Current`.

```
namespace System.Collections.Generic
{
  public interface IEnumerator<out T> :
                                IEnumerator
  {
    T Current { get; }
  }
}
```

```
namespace System.Collections
{
  public interface IEnumerator
  {
    object Current { get; }
    bool MoveNext();
    void Reset();
  }
}
```

Listado 11-21 Interface IEnumerator<T> e IEnumerator

Note que la diferencia entre `IEnumerator<T>` e `IEnumerator` es que en el primer caso la propiedad `Current` es de tipo `T` y en el caso de `IEnumerator` es tipo `object`.

El esquema típico de utilización es

```
IEnumerator<A> enumerador =
...objeto de un tipo que implemente IEnumerator<A> ...
while (enumerador.MoveNext()){
...Procesar enumerador.Current ...
}
```

1) El intento de aplicar `Current` si no hay elementos para dar (aún no hemos hecho un `MoveNext` que nos haya devuelto true, o el último `MoveNext` nos dio false) provoca una excepción.
2) Se puede aplicar `Current` tantas veces como se quiera; éste siempre devolverá el mismo valor si no se ha hecho `MoveNext`.
3) Si se hace `Reset` se vuelve a poner el enumerador en la misma posición de antes de hacer el primer `MoveNext`.

11.6.2 Enumerador reverso de un array

La clase del Listado 11-22 implementa un enumerador `Reverso<T>` que a partir de un array nos da los elementos del array recorridos desde el final hacia el principio. Si se tiene el array

```
int[] a = new int[]{10, 20, 30, 40};
```

y se hace

```
IEnumerator<int> r = new Reverso<int>(a);
while (r.MoveNext())
Console.Write(" {0}", r.Current);
```

se escribirá

```
40  30  20  10
```

```
public class Reverso<T> : IEnumerator<T>
{
  T[] items; int cursor;
  bool hayCurrent;
  public Reverso(T[] items)
```

```
  {
    this.items = items;
    cursor = items.Length;
  }
  public bool MoveNext()
  {
    cursor--;
    hayCurrent = (cursor >= 0);
    return hayCurrent;
  }
  public T Current{
    get
    {
      if (hayCurrent) return items[cursor];
      else
        throw new InvalidOperationException("Enumerador fuera de posición");
    }
  }
  object IEnumerator.Current
  {
    get { return Current; }
  }
  public void Reset()
  {
    cursor = items.Length; hayCurrent = false;
  }
  public void Dispose(){}
}
```

Listado 11-22 Una implementación de IEnumerator<int> para Reverso

La interface `IEnumerator<T>` hereda de

public interface IEnumerator<out T> : IDisposable, IEnumerator

La interface `IDisposable` tiene un método `void Dispose()`, de modo que quien implemente `IEnumerator<T>` debe también implementar el método `Dispose`. Este concepto de `IDisposable` no hace falta para este ejemplo, y por ello por ahora en esta implementación de `Dispose` no se hace nada.

Note que como la interface `IEnumerator<T>` hereda de `IEnumerator` hay que proporcionar entonces una implementación explícita de la propiedad `Current` de `IEnumerator`. En este caso lo que se hace es devolver el mismo objeto que el `Current` de `IEnumerator<T>`, pero considerado solo como de tipo `object`.

Lamentablemente, la primera versión de C# no incluía aún la genericidad. De modo que inicialmente interfaces como `IEnumerator` e `IEnumerable` no existían en su versión genérica `IEnumerator<T>` e `IEnumerable<T>`. Al introducirse la genericidad, para mantener la compatibilidad hacia atrás, Microsoft diseñó que las nuevas interfaces genéricas heredasen de sus versiones no genéricas. De modo que los nuevos objetos de los tipos genéricos pudiesen usarse en códigos escritos antes de la existencia de la genericidad sin tener que hacer modificaciones en estos últimos (cumpliendo así con el principio abierto-cerrado que se estudia en el capítulo 16).

11.6.3 Enumerador de pares en un intervalo

En este ejemplo del enumerador `Reverso` la fuente original de los datos es un array, es decir, que dichos datos están físicamente en memoria. En este caso el enumerador lo que expresa es una "maquinaria" diferente para recorrer los datos del array. El Listado 11-23 nos muestra un enumerador `ParesEnumerator` que nos da los números pares comprendidos en un determinado intervalo. Observe que los números que el enumerador va devolviendo no están guardados físicamente en memoria, sino que se van calculando según se demanden por el código que esté usando al enumerador, de modo que podemos considerar que este enumerador expresa una suerte de "colección virtual de datos".

```
class ParesEnumerator : IEnumerator<int>
{
  public int Min { get; private set; }
  public int Max { get; private set; }
  int par; bool hayCurrent;
  public ParesEnumerator(int min, int max)
  {
    Min = min; Max = max;
    Reset();
  }
  public bool MoveNext()
  {
    par += 2;
    hayCurrent = par <= Max;
    return hayCurrent;
  }
  public int Current
  {
    get
    {
      if (hayCurrent) return par;
      else
        throw new InvalidOperationException("Enumerador fuera de posición");
    }
  }
  object IEnumerator.Current
  {
    get
    {
      return Current;
    }
  }
  public void Reset()
  {
    if (Min % 2 == 0) par = Min - 2;
    else par = Min - 1;
    hayCurrent = false;
  }
  public void Dispose() { }
}
```

Listado 11-23 Enumerador ParesEnumerator en un Intervalo

11.6.4 Un enumerador de un enumerador

Tenga presente que un enumerador puede ser infinito, y mientras `MoveNext` retorne true podemos estar aplicando `Current`. El Listado 11-24 muestra un ejemplo trivial de un enumerador infinito: siempre retorna un mismo elemento.

```
class EumeradorInfinito : IEnumerator<int>
{
  public bool MoveNext() { return true; }
  public int Current { get { return 0; } }
  object IEnumerator.Current { get { return 0; } }
  public void Reset() { }
  public void Dispose() { }
}
```

Listado 11-24 Enumerador Infinito

Es frecuente aplicar un enumerador al resultado de un enumerador. Uno de los más simples de estos enumeradores es un enumerador que devuelve los *n* primeros elementos de otro enumerador. Por analogía con la terminología utilizada por el propio .NET, llamaremos `Take` a este método que nos proporciona un tal enumerador. El código del Listado 11-25 implementa un tal método `Take` que retorna un objeto del tipo `TakeEnumerator`. Note que el método `MoveNext` de `TakeEnumerator` se ha implementado de modo tal que si el enumerador original tiene más de *n* elementos solo se devuelven *n*, y si tiene menos de *n* pues devuelve `false` para que se termine cuando se acaben los del origen. La propiedad `Current` solo tiene que limitarse entonces a devolver el mismo `Current` que devolvería el enumerador de origen.

```
class Iteradores
{
  public static IEnumerator<T> Take<T>(IEnumerator<T> fuente, int n)
  {
    return new TakeEnumerator<T>(fuente, n);
  }
  class TakeEnumerator<T>: IEnumerator<T>
  {
    int n;
    IEnumerator<T> fuente;
    int k;
    bool hayCurrent;
    public TakeEnumerator(IEnumerator<T> fuente, int n)
    {
      this.n = n;
      this.fuente = fuente;
    }
    public bool MoveNext()
    {
      if (k < n)
      {
        k++;
        return hayCurrent = fuente.MoveNext();
      }
      else
        return hayCurrent = false;
```

```
    }
    public T Current
    {
      get
      {
        if (hayCurrent) return fuente.Current;
        else
          throw new InvalidOperationException("El enumerador está fuera de
                                               posición");
      }
    }
    object IEnumerator.Current { get { return Current; } }
    public void Reset()
    {
      k = 0; fuente.Reset();
    }
    public void Dispose() { }
  }
}
```

Listado 11-25 Enumerador que devuelve los n primeros elementos de otro enumerador

La ejecución del código del Listado 11-26 escribe las líneas

```
10 12 14
10 12 14 16 18 20
```

En primer caso, aunque el enumerador fuente tiene más de 3 elementos a `Take` solo se le ha pedido los 3 primeros. En el segundo caso, aunque se piden los 10 primeros solo se devuelven 6 elementos ya que el enumerador original solo tiene 6.

Note que entre ambos recorridos se ha hecho `pares.Reset()` porque el primer recorrido por el enumerador `pares` quedó en la posición siguiente al valor `14`. Si no se hubiese hecho `Reset`, el segundo recorrido hubiese escrito solo los valores `16 18 20`

```
IEnumerator<int> pares = new ParesEnumerator(10, 20);
IEnumerator<int> pares1 = Enumeradores.Take<int>(pares, 3);
while (pares1.MoveNext())
{
  Console.Write("  {0}", pares1.Current);
}
Console.WriteLine();

pares.Reset();

IEnumerator<int> pares2 = Enumeradores.Take<int>(pares, 10);
while (pares2.MoveNext())
{
  Console.Write("  {0}", pares2.Current);
}
Console.WriteLine();
```

Listado 11-26 El método Take aplicado al enumerador ParesEnumerator

11.6.5 Arrays e IEnumerator

A los arrays se les puede aplicar un método `GetEnumerator` que devuelve un `IEnumerator` (Listado 11-27)

```
int[] decenas = new int[] { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
IEnumerator e = decenas.GetEnumerator();
Console.WriteLine("Recorriendo el array con MoveNext");
while (e.MoveNext())
{
  Console.Write("  {0}", e.Current);
  //Console.Write("  {0}", e.Current + 1);
}
```

Listado 11-27 Recorriendo el array con IEnumerator

Este `GetEnumerator` devuelve un `IEnumerator` y recuerde que en este caso `Current` devuelve `object` aunque el valor real que esté devolviendo sea en este caso `int`. Lo que ocurre es que `Write` escribe el valor de este `int`. Si se quitara el comentario a la línea `Console.Write(" {0}", e.Current + 1);` tendríamos error de compilación.

Aunque `decenas` es un array de `int`, lamentablemente C# da error de compilación ante el intento de escribir

IEnumerator<int> e = decenas.GetEnumerator();

Para obviar este error habría que escribir una operación de *casting* explícita (Listado 11-28)

```
IEnumerator<int> eint = (IEnumerator<int>)(decenas.GetEnumerator());
while (eint.MoveNext())
{
  Console.Write("  {0}", eint.Current);
  //La linea siguiente NO ES ERROR de compilación
  Console.Write("  {0}", eint.Current + 1);
}
```

Listado 11-28 Recorriendo el array con IEnumerator<int>

11.6.6Interfaces IEnumerable<T> e IEnumerable. Ciclo foreach

El Listado 11-29 nos muestra la versión genérica y la no genérica de `IEnumerable`. Una clase que implemente `IEnumerable` lo que tiene que hacer es implementar un método `GetEnumerator` que devuelva un `IEnumerator` (ídem para su correspondiente genérico).

```
namespace System.Collections.Generic
{
  public interface IEnumerable<out T> :
                                IEnumerable
  {
    IEnumerator<T> GetEnumerator();
  }
}
```

```
namespace System.Collections
{
  public interface IEnumerable
  {
    IEnumerator GetEnumerator();
  }
```

Listado 11-29 Interfaces IEnumerable<T> e IEnumerable

Si simbólicamente el concepto de **enumerador** está asociado al de un "motor que permite recorrer una secuencia de datos", el concepto de **enumerable** está asociado al de la "propia secuencia como tal".

C# ofrece una azúcar sintáctica para recorrer un `IEnumerable` mediante la construcción de ciclo `foreach`.

Si `coleccion` es de tipo `IEnumerable<T>`, entonces esta colección se puede recorrer con una construcción de la forma

```
foreach (T x in coleccion) {... acción a realizar con x ...}
```

Y si `coleccion` es `IEnumerable` entonces se puede recorrer con

```
foreach (object x in coleccion) {... acción a realizar con x ...}
```

Note que ésta es una forma más declarativa de recorrer una colección que hacerlo explícitamente mediante `MoveNext` y `Current` y que nos previene de cometer el error de aplicar `Current` cuando no estamos adecuadamente posicionados en el iterador. Es el compilador de C# quien convierte el código del `foreach` anterior a su variante con enumerador

```
IEnumerator<T> enumerador = coleccion.GetEnumerator();
while (enumerador.MoveNext())
{ T x = enumerador.Current;
   ...acción a realizar con x...
 }
```

Es una práctica muy utilizada que una clase que quiera expresar a una "colección de datos" implemente la interface `IEnumerable<T>` (o `IEnumerable`) y oculte al código que implementa el enumerador dentro de una clase interna a ésta. En lugar de ofrecer directamente la clase `ParesEnumerator` (Listado 11-23) podría ofrecerse una clase `Pares` como la del Listado 11-30.

```
public class Pares : IEnumerable<int>
{
  public int Min { get; private set; }
  public int Max { get; private set; }
  public Pares(int min, int max)
  {
    Min = min; Max = max;
  }

  public IEnumerator<int> GetEnumerator() {
    return new ParesEnumerator(Min, Max);
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }

  //...DEFINICIÓN DE LA CLASE ParesEnumerator DEL LISTADO 11.23...
}
```

Listado 11-30 Pares en un intervalo como un IEnumerable<T>

Ahora si disponemos de esta clase podemos escribir directamente

```
foreach (int k in new Pares(10, 20))
      Console.Write("  {0}", k);
```

y obtener como respuesta

```
10   12  14  16  18  20
```

Realmente basta con que una clase tenga un método `GetEnumerator` que devuelva un `IEnumerator<T>` para que el compilador de C# acepte un código como el del anterior `foreach` sin necesidad de que la clase diga que implementa la interface `IEnumerable<int>`. Sin embargo, en tal caso intentar hacer la asignación `IEnumerable<int> ie = new Pares(10, 20)` provoca error de compilación).

11.6.7 Arrays e IEnumerable

Aunque los arrays son tipos predefinidos integrados al lenguaje, C# también considera a los arrays como que implementan implícitamente la interface `IEnumerable`, de modo que el código del Listado 11-31 es correcto

```
int[] decenas = new int[] { 10, 20, 30, 40, 50, 60, 70, 80, 90 };
foreach (int k in decenas)
  Console.WriteLine(k);
Console.WriteLine();
foreach (object x in decenas)
  Console.WriteLine(x);
IEnumerator ie = decenas.GetEnumerator();
while (ie.MoveNext())
  {
    Console.WriteLine(ie.Current);
  }
```

Listado 11-31 Recorriendo un array con foreach como IEnumerable

El código siguiente también es correcto porque array hace una implementación explícita de la interface `IEnumerable<T>`

```
IEnumerator<Color> ieColores1 = ((IEnumerable<Color>)colores).GetEnumerator();
while (ieColores1.MoveNext()){ Console.WriteLine(ieColores1.Current); }
```

Sin embargo, el siguiente código da error de compilación porque la implementación implícita que hace array de `GetEunumerator` es la de `IEnumerable`

```
IEnumerator<Color> ieColores2 = colores.GetEnumerator();
```

El lector puede pensar, y con razón, que ya que los arrays son una suerte de construcción genérica predefinida, y la implementación implícita que hacen los arrays debiera ser la de `IEnumerable<T>`. Sin embargo, los arrays forman parte del lenguaje desde la primera versión de C#, cuando se tenían las interfaces `IEnumerable` e `IEnumerator` pero no existían aún las interfaces genéricas.

11.7 Iteradores con operador yield

11.7.1 Una clase Intervalo con diferentes iteradores

La clase `Pares` del Listado 11-30 define un objeto que es en sí mismo un `IEnumerable<int>` que representa a los pares de un intervalo. Un mejor diseño sería tal vez el de ofrecer una clase `Intervalo` como la del Listado 11-32 y que sea un objeto de tipo `Intervalo` el que pueda tener diversas formas de recorrido. El `Intervalo` cuyo esquema se muestra en el Listado 11-32 tiene en este caso tres formas de recorrido, y el `Intervalo` en sí mismo es `IEnumerable<int>`, lo que en este caso permite recorrer todos los elementos del intervalo. La clase ofrece también dos propiedades `Pares` e `Impares` que a su vez devuelven también `IEnumerable<int>` para recorrer respectivamente los valores pares y los valores impares del intervalo.

```
public class Intervalo: IEnumerable<int>
{
  public int Min { get; private set; }
  public int Max { get; private set; }
  public Intervalo(int inf, int sup) //...

  public IEnumerator<int> GetEnumerator() //...

  IEnumerator IEnumerable.GetEnumerator() //...

  public IEnumerable<int> Pares //...

  public IEnumerable<int> Impares //...

  //...Otros métodos y propiedades de Intervalo
}
```

Listado 11-32 Una clase Intervalo con varios iteradores

Para completar la implementación de esta clase `Intervalo` habría que implementar una "maquinaria" `IEnumerator<int>` para recorrer en este caso todos los valores del intervalo y definir a su vez dos implementaciones de `IEnumerable<int>` para que las propiedades `Pares` e `Impares` devuelvan instancias de estas clases. Esto sería un trabajo bastante laborioso que implicaría implementar los correspondientes `IEnumerator<int>` con las respectivas maquinarias de recorrido con `MoveNext` dejamos al lector este trabajo para que le sirva de entrenamiento. Felizmente, C# ofrece, mediante el operador `yield`, un recurso que hace más simple y cómodo este tipo de implementación, como se muestra en el Listado 11-33.

```
public class Intervalo: IEnumerable<int>
{
  public int Min { get; private set; }
  public int Max { get; private set; }
  public Intervalo(int inf, int sup)
  {
    Inf = inf; Sup = sup;
  }

  public IEnumerator<int> GetEnumerator() {
    for (int k = Inf; k <= Sup; k++) yield return k;
  }

  IEnumerator IEnumerable.GetEnumerator()
  {
    return GetEnumerator();
  }

  public IEnumerable<int> Pares {
    get
    {
      int par;
      if (Inf % 2 == 0) par = Inf;
      else par = Inf + 1;
      for (int k = par; k <= Sup; k+=2)
        yield return k;
    }
  }

  public IEnumerable<int> Impares
  {
    get
    {
      int impar;
      if (Inf % 2 == 0) impar = Inf+1;
      else impar = Inf;
      for (int k = impar; k <= Sup; k += 2)
        yield return k;
    }
  }
}
```

Listado 11-33 Utilización del operador yield para implementar iteradores

El mecanismo del `yield return` trabaja asociado al ciclo `foreach`. Cuando se hace

```
foreach (int x in intervalo.Pares) Console.Write("  {0}", x);
```

antes de intentar hacer la primera iteración se invoca por primera vez a la parte `get` de la propiedad `intervalo.Pares`. Cuando la ejecución de la propiedad ejecuta el código `yield return k`, el valor `k` se devuelve como valor a la variable de iteración del `foreach` (en este caso x). Cuando el `foreach` intenta hacer otra iteración, entonces se continúa la ejecución en la instrucción siguiente a la del último `yield return`. La iteración del `foreach` termina cuando al invocar a la continuación de la ejecución del método (propiedad) éste a su vez termina.

11.7.2Operador yield break

La ejecución del método puede abortarse también ejecutando `yield break`, lo cual a su vez aborta la ejecución del correspondiente `foreach`. El código del Listado 11-34 nos da un iterador vacío.

```
public IEnumerable<int> Vacio
{
  get { yield break; }
}
```

Listado 11-34 Iterador vacío

11.7.3Iterador infinito para los números de Fibonacci

En el código del iterador se puede tener más de un `yield return`, como se muestra en el iterador `Fibonacci` del Listado 11-35. Note que este iterador es infinito, y siempre estará dando un número de la secuencia infinita de números de Fibonacci[2].

```
public class Iteradores
{
  public static IEnumerable<long> Fibonacci
  {
    get
    {
      long penultimo = 1, ultimo = 1;
      yield return penultimo;
      yield return ultimo;
      while (true)
      {
        long temp = penultimo;
        penultimo = ultimo;
        yield return ultimo += temp;
      }
    }
  }
}
```

Listado 11-35 Iterador con más de un yield

11.7.4 Combinación de IEnumerables

Utilizando `yield` es más simple implementar iteradores a partir de otros iteradores. Un método como el `Take` del Listado 11-25 ahora se podría implementar de forma sencilla (Listado 11-36).

```
public class Iteradores
{
  //...
  public static IEnumerable<T> Take<T>(IEnumerable<T> fuente, int n)
  {
    int k = 0;
    foreach (T x in fuente)
```

[2] Realmente en este ejemplo el intento de infinitud terminará dando resultados erróneos, porque en algún momento se desbordará la representación de un número long.

```
    {
      if (k < n)
      {
        yield return x;
        k++;
      }
      else yield break;
    }
  }
}
```

Listado 11-36 Nueva implementación del método Take ahora usando yield

Note cómo se ha utilizado el `yield break` para abortar la iteración cuando ya se han obtenido *n* elementos. Si la fuente tiene menos de *n* elementos, la iteración del `Take` terminará porque termina la iteración del `foreach` de la fuente.

El Listado 11-37 nos muestra un iterador `Concat` que es la concatenación de dos enumerables: primero se regresan los elementos del primer enumerable y luego los del segundo enumerable.

```
public static IEnumerable<T> Concat<T>(IEnumerable<T> enum1,
                                       IEnumerable<T> enum2)
{
   foreach (T x in enum1) yield return x;
   foreach (T y in enum2) yield return y;
}
```

Listado 11-37 Concatenación de dos enumerables

NET ofrece una clase `Enumerable` en la que hay una gran variedad de métodos que reciben enumerables para devolver otros enumerables.

11.7.5 La naturaleza lazy de los enumerables

El operador `yield` facilita la implementación de iteradores perezosos[3] en los que un enumerable podrá representar a una colección de datos, pero no se quiere tener que formar toda la colección (que de hecho podría ser conceptualmente infinita) para irla procesando. De ahí el término perezoso (lazy), porque el próximo elemento de la colección no se "produce" hasta que el código que lo procesa lo demanda.

Si queremos un método `Ordena` que reciba un array de elementos y los devuelva en un enumerable, podría hacerse una implementación como la del Listado 11-38. Note que esta implementación tiene el inconveniente de que el código que procese el resultado de este método no podrá disponer del primer elemento hasta que todo el array se haya ordenado.

```
public static IEnumerable<T> OrdenaNoLazy<T>(T[] elems) where T : IComparable<T>
```

[3] En inglés lazy

```
{
  for (int k = 0; k < elems.Length - 1; k++)
    for (int j = k + 1; j < elems.Length; j++)
      if (elems[j].CompareTo(elems[k])<0) {
        T temp = elems[j];
        elems[j] = elems[k];
        elems[k] = temp;
      }
  return elems;
}
```

Listado 11-38 Iterador de ordenación no perezoso

Si la cantidad de elementos del array es muy grande, el costo de esta espera puede no ser deseado por el código que procesará el resultado de la ordenación. Imagine que se ordenan 100,000 elementos; habría una espera proporcional a $100{,}000^2$ antes de que se procese el primer elemento, cuando tal vez se quieran solo los 100 primeros para decidir no seguir procesando[4].

Si el método `Ordena` se implementase como en el Listado 11-39 se devolverá el primer elemento de la ordenación cuando haya transcurrido solo un tiempo proporcional a 100,000!

```
public static IEnumerable<T> OrdenaLazy<T>(T[] elems) where T : IComparable<T>
{
  for (int k = 0; k < elems.Length - 1; k++)
  {
    for (int j = k + 1; j < elems.Length; j++)
      if (elems[j].CompareTo(elems[k]) < 0)
      {
        T temp = elems[j];
        elems[j] = elems[k];
        elems[k] = temp;
      }
    yield return elems[k];
  }
}
```

Listado 11-39 Ordenación perezosa (lazy)

Este método combinaría perfectamente con el método `Take` visto anteriormente. Pruebe a medir el tiempo de ejecutar el código del Listado 11-40

```
int[] a = new int[100000];
//...
Stopwatch crono = new Stopwatch();
crono.Start();
foreach (int k in Iteradores.Take(Iteradores.OrdenaLazy(a), 10)) ;
Console.WriteLine(crono.ElapsedMilliseconds);
crono.Restart();
```

[4] Si esto forma parte de una interfaz visual (GUI) de interacción con el usuario, éste puede sentirse inconforme (o desesperado) con la espera cuando tal vez aún no ha visto aún ningún resultado.

```
foreach (int k in Iteradores.Take(Iteradores.OrdenaNoLazy(a), 10)) ;
Console.WriteLine(crono.ElapsedMilliseconds);
```

Listado 11-40 Ejecutando un iterador no perezoso y uno perezoso

No se confunda con el uso que a veces se le da al término perezoso en el lenguaje coloquial. Aquí no quiere decir que trabaje lento, sino que trabaja cuando no se puede postergar más porque el código que lo usa le está reclamando un dato. En la práctica es el que procesa los datos (y no el que los produce) quien puede decidir si terminar antes de procesarlos todos.

11.7.6 Iteradores y recursividad

El uso del operador `yield` encaja a la perfección con la recursividad. El iterador del Listado 11-41 nos da un enumerable con los movimientos de las Torres de Hanoi (ver Capítulo 8 Recursividad) sin tener que guardar los movimientos a realizar en una estructura de datos.

```
class Iteradores
{
  //...
  public static IEnumerable<string> TorresDeHanoi(int n, string origen,
                                          string auxiliar, string destino)
  {
    if (n < 1) yield break;
    if (n == 1) yield return "De " + origen + " a " + destino;
    else
    {
      foreach (string s in TorresDeHanoi(n - 1, origen, destino, auxiliar))
        yield return s;
      yield return "De " + origen + " a " + destino;
      foreach (string s in TorresDeHanoi(n - 1, auxiliar, origen, destino))
        yield return s;
    }
  }
}
```

Listado 11-41 Iterador con los movimientos de las Torres de Hanoi

Note que el `IEnumerable` que es `TorresDeHanoi` se basa a su vez en recorrer el `IEnumerable` que devuelve recursivamente `TorresDeHanoi`. Pruebe a implementar este método sin usar el operador `yield`; lograr el efecto de la recursividad con la maquinaria del `MoveNext` del `IEnumerator` es bastante complicado y eso es lo que por suerte nos hace el compilador de C# tras el telón.

Es muy interesante estudiar la maquinaria que se implementa por detrás para hacer posible el funcionamiento del `yield` integrado con el `foreach`, pero esto se sale del alcance de este libro. Recomendamos al lector interesado ver la documentación publicada en MSDN.

11.8 Interfaces sobre estructuras de datos

En el Capítulo 13 Estructuras de Datos se estudian varias de las estructuras de datos más importantes para la Ciencia de la Computación como listas, pilas, colas, diccionarios y árboles. Aunque por supuesto que no es intención de este libro estudiar todas las interfaces que hay en las bibliotecas de .NET, hay dos interfaces que son de interés analizar aquí por la abstracción que representan para expresar funcionalidades comunes a las diferentes estructuras de datos. Estas son las interfaces `ICollection` e `IList`.

11.8.1 Interface ICollection

La interface `ICollection` (Listado 11-42) representa la forma más abstracta de expresar una colección de datos con independencia de cómo se implemente ésta.

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }
  bool Contains(T item);
  bool IsReadOnly { get; }
  void Add(T item);
  bool Remove(T item);
  void Clear();
  void CopyTo(T[] array, int arrayIndex);
}
```

Listado 11-42 Interface ICollection

Inspirados por los conocimientos de teoría de conjuntos que podemos tener de las matemáticas, las tres cualidades más elementales básicas que debiera tener toda colección de elementos son:

1. *Cantidad.* Poder conocer cuántos elementos tiene. Lo que se expresa aquí por la propiedad `Count` (note que obviamente esta propiedad es de solo lectura, porque no tiene sentido cambiar directamente este valor).
2. *Pertencia.* Saber si un elemento está contenido en la colección. Lo que se expresa aquí por el método `bool Contains(T item)`, que devuelve `true` o `false` en dependencia de si `item` está en la colección.
3. *Poder conocer (tener acceso) a todos los elementos de la colección.* Lo que se expresa aquí en la forma más primaria posible y es que se puedan recorrer. Por eso se considera que `ICollection` debe ser también `IEnumerable<T>` e `IEnumerable`. Esto permite hacer `foreach(T x in c)` y `foreach(object x in c)`.

Debe destacarse la naturaleza minimalista anterior. Si usando una determinada implementación (que quien trabaja con la colección no tiene por qué conocer) se forma un objeto de tipo `ICollection`, entonces al menos las tres funcionalidades anteriores deberían poder aplicarse. Para qué nos interesaría "una colección" si al menos no podemos saber cuántos elementos la forman, tenemos alguna forma de poder acceder a los mismos y no podemos saber si un determinado elemento está contenido en la colección.

Pensando minimalistamente, se pudiera decir que si `ICollection` es `IEnumerable` entonces no haría falta tener una propiedad `Count` porque estos pudieran implementarse en la forma

```
int count=0;
foreach (T x in c) count++;
return count;
```

Tampoco tendría que tener un método `Contains` que podría implementarse con

```
foreach (T x in c)
 if (x.Equals(item)) return true;
return false;
```

Pero ¿qué pasaría entonces si el enumerable es infinito? `Count` nunca devolvería un valor y `Contains` podría entonces nunca devolver `false`. Al colocar ambas funcionalidades dentro de la interface `IEnumerable` se pasa la responsabilidad de tomar la decisión sobre esto a quien haga la implementación.

La propiedad `IsReadOnly` permite expresar si la colección puede ser modificada. De este modo se podría implementar un tipo de `IEnumerable` tal, que una vez creada una instancia esta no pueda ser modificada.

Las tres operaciones `Add`, `Remove` y `Clear` pueden ser aplicadas solo si el valor de la propiedad `IsReadOnly` es true; de lo contrario deben lanzar una excepción. `Add` añade un elemento a la colección, `Remove` quita un elemento de la colección y `Clear` "vacía la colección", es decir, deja la colección sin elementos.

Por último el método `CopyTo` copia los elementos de la colección hacia un array a partir de una posición en el array destino. Debe reportar excepción si el índice se sale de los rangos del array o si la cantidad de elementos de la colección se salen del espacio indicado en el array

Se puede pensar, y con razón, que el método `CopyTo` desentona en comparación con los restantes de la interface `IEnumerable`. La intención detrás de su inclusión en esta interface debió ser la de proporcionar una manera uniforme y eficiente de copiar "en memoria" los datos de la colección (recuerde que por ser `IEnumerable` una colección no tiene por qué tener los datos físicamente guardados en memoria, sino que los puede producir por demanda).

11.8.2 Interface IList

Una funcionalidad más específica de una colección de datos es la de que cada elemento tenga una "posición" dentro de la colección. La interface `IList` (Listado 11-43) expresa esta funcionalidad. Note que un `IList` hereda de `ICollection` y de `IEnumerable` lo que nos permite usar un objeto de tipo `IList` en un código en donde no interese la posición de los elementos. Pero si nos interesa la posición, entonces también se dispone de un indizador.

```
namespace System.Collections.Generic
{
```

```
  public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
  {
    T this[int index] { get; set; }
    void Insert(int index, T item);
    void RemoveAt(int index);
    int IndexOf(T item);
  }
}
```

Listado 11-43 Interface IList

Lo más distintivo de `IList<T>` es el indizador

```
T this[int index] { get; set; }
```

De modo que el código del Listado 11-44 es sintácticamente correcto

```
IList<string> lista;
int k=0,j=0;
//...
lista[k] = "azul";
Console.WriteLine(lista[j]);
```

Listado 11-44 Usando los indizadores en IList

Hay varios aspectos a destacar aquí. El uso del indizador no entra en contradicción con que la propiedad `IsReadOnly` (que se hereda de `IEnumerable`) devuelva `true`; si hacemos `lista[k] = "azul"` no estamos cambiando la `lista`, sino cambiando el valor que está en la posición `k`. Como `IList` proviene de `IEnumerable` tiene también entonces la propiedad `Count`, lo que permite asumir como convenio (de modo parecido a los arrays) que las posiciones válidas son aquellas entre `0` y `Count-1`. De modo que el indizador nos permite dar el efecto de consultar o modificar los valores que están en las distintas posiciones de la colección, dando excepción si la supuesta posición se sale del rango anterior. Pero a través del indizador no se pueden añadir o eliminar valores y sus posiciones.

Recuerde que `IList` es `IEnumerable`, y que por ser `IEnumerable` tiene el método añadir `Add`, de modo que usando `Add` un objeto de tipo `IList` puede "crecer" y por tanto añadir y crear valores en nuevas posiciones. Las operaciones propias para añadir y quitar elementos deben estar entonces asociadas a alguna posición o índice. El método `void Insert(int index, T item)` inserta el elemento `item` en la posición `index` (que por supuesto debe estar dentro del rango válido), lo que automáticamente desplaza en una posición los elementos que estaban a partir de la posición `index` en adelante, aumentando el `Count` de la colección. Por el contrario, el método `void RemoveAt(int index)` quita de la colección el elemento que está en la posición `index` y automáticante desplaza a la posición anterior a los elementos que están a partir de `index`, decrementando el valor de `Count`.

La forma en que han sido diseñadas las interfaces `IEnumerable`, `ICollection` e `IList` son un buen ejemplo de que en su concepción se han seguido los principios SOLID de responsabilidad única, abierto-cerrado, segregación de interfaces y sustitución de Liskov, los cuales se estudian en el Capítulo 16.

El método `int IndexOf(T item)` da el valor de la posición en la que se encuentra el valor `item`. El convenio a adoptar es que si `item` no se encuentra en la lista entonces se devuelva `-1`. La interface `IList` no se pronuncia sobre cuál valor debe devolver si `item` se encuentra en más de una posición.

En muchas implementaciones de `IList`, lo que devuelve el método `IndexOf` es la posición menor (es decir, la más cercana a 0) en la que se encuentra el elemento, pero eso no es un requerimiento y por tanto cuando usted use un `IList` no debe suponer esto.

Por convenio, los arrays son considerados de tipo `IList`. La asignación `lista = a` del Listado 11-45 es correcta. Su ejecución daría el resultado que se muestra en la Figura 11-1Figura 11-2

```
int[] a = new int[] {2, 4, 6, 8};
IList<int> lista = a;
for (int k = 0; k < list.Count; k++)
  Console.Write("  {0}", lista[k]);
Console.WriteLine();
lista.RemoveAt(2);
```

Listado 11-45 Array considerado como IList

```
2  4  6  8

Unhandled Exception: System.NotSupportedException; Collection was of
a fixed size
```

Figura 11-2 Usando un array a través de un IList

El array `a` se puede asignar a la variable `lista` porque un array de enteros `int[]` se considera un `IList<int>`, y por lo tanto recorrer los elementos de `lista` es válido. El acceso a los elementos de `lista` mediante el indizador equivale a acceder a los elementos del array y la cantidad `Count` es la longitud `Length` del array. Sin embargo, el intento de hacer `lista.RemoveAt(2)` provoca la excepción `NotSupportedException` porque un array, si bien es una colección indizable, es de tamaño fijo, por lo que no se le pueden quitar ni añadir elementos.

Si c es de tipo `IList<T>` , los siguientes recorridos de c deben dar los valores en el mismo orden

```
foreach (T x in c) …hacer algo con x…
for (int i=0; i<c.Count; i++) …hacer algo con c[i]…
```

Si bien esto se cumple por la forma en que en C# se implementan los array y porque un array `T[]` se considera como de tipo `IList<T>`, nada hay que garantice que se cumpla para cualquier otra implementación que se haga de `IList`. De modo que usted no debe programar un código que al usar un `IList` dependa de que esto sea diferente o no.

11.9 Interfaces vs Clases Abstractas

Un patrón adecuado de programación cuando se va a implementar una funcionalidad es basar esta lo más posible en el uso de interfaces. Será el código que vaya a usar dicha funcionalidad el que tenga que decidir qué implementación de la interface es la que le conviene utilizar. Observe las dos implementaciones de un método `Ordenar` del Listado 11-46

```
static void Ordenar(IComparable[] items)
{
  for (int k = 0; k < items.Length - 1;
       k++)
    for (int j = k + 1; j < items.Length;
         j++)
      if (items[j].CompareTo(items[k])<0)
      {
        IComparable temp = items[j];
        items[j] = items[k];
        items[k] = temp;
      }
}
```

```
static void Ordenar(IList<IComparable>
                    items)
{
  for (int k = 0; k < items.Length - 1; k++)
    for (int j = k + 1; j < items.Length;
         j++)
      if (items[j].CompareTo(items[k])<0)
      {
        IComparable temp = items[j];
        items[j] = items[k];
        items[k] = temp;
      }
}
```

Listado 11-46 Ordenar basado en array y Ordenar basado en IList

La primera implementación solo vale para ordenar un array, mientras que la segunda implementación vale para ordenar cualquier colección de un tipo implemente la interface `IList`, lo que no excluye que dicha colección pueda ser un array.

Usted puede considerar, y con razón, que una clase totalmente abstracta, es decir que no tenga variables de instancia y en la que todos los métodos y propiedades sean abstractos, podría ser equivalente a una interface. Los beneficios del ejemplo anterior podrían lograrse también si `AbstractList` fuese la clase abstracta del Listado 11-47

```
abstract class AbstractList<T>
{
  public abstract T this[int index] { get; set; }
```

```
  public abstract int Count { get; }
  public abstract void Insert(int index, T item);
  public abstract void RemoveAt(int index);
  public abstract int IndexOf(T item);
}
```

Listado 11-47 Clase abstracta para Lista

Pero una clase abstracta puede tener algo de implementación. El objetivo de la inclusión del concepto de interface, con independencia del de clase abstracta, es facilitarle al programador, al compilador y al framework de ejecución, la distinción de una suerte de tipo en "estado puro", es decir, la descripción de una funcionalidad que no se compromete con ninguna forma de implementación (salvo la signatura que tienen los métodos y las propiedades).

Como se vio anteriormente en este capítulo, tiene utilidad que una clase implemente varias interfaces con independencia de que además herede de otra clase. Esto se puede interpretar como diferentes rasgos o funcionalidades que puede tener un objeto instancia de dicha clase. Esto no podría lograrse solo con clases abstractas porque no se puede heredar de más de una clase.

Hay lenguajes de programación, como C++ y Eiffel, que no tienen interfaces, sino solo clases abstractas, pero que permiten la herencia múltiple, es decir que una clase pueda heredar de más de una clase base. En tales casos, si por un lado se pueden tener clases totalmente abstractas, y por otro lado se puede heredar de varias clases, se podrían obtener los mismos beneficios que se han expuesto en este capítulo con las interfaces.

Se sale de los objetivos de este libro una discusión sobre este tema, pero lo cierto es que si esto se permite entonces el modelo de representación en memoria de los objetos, y la forma de implementar el polimorfismo, se hacen bastante complicados. Algunos lenguajes que no tienen herencia múltiple, pero que promueven el uso de interfaces, como son los casos de Java y C#, han considerado que son más las complicaciones que los beneficios y se evitan el problema no permitiendo herencia múltiple.

11.10 Poniendo código a una interface con los Métodos Extensores

Los métodos extensores que se introdujeron en al Capítulo 10 vienen como anillo al dedo para lograr el efecto de poner “implementaciones en una interface”. Retome el método `Take` del Listado 11-36, el cual se implementó como un método estático dentro de una clase `Iteradores`. Fíjese que tal implementación no depende para nada de cómo es la implementación del tipo `IEnumerable<T>` que haga el objeto que se pasa como parámetro `fuente`. Note que independientemente de la implementación que tenga `IEnumerable` en este caso, `Take` podría aplicarse lo mismo a un `IEnumerable` vacío (retornaría en ese caso un `IEnumerable` vacío) que a un `IEnumerable` infinito (porque terminaría al devolver los primeros `k` elementos). De modo que pudiera aspirarse a que

una tal implementación de `Take` pudiese formar parte de la propia definición de la interface `IEnumerable`, por ejemplo como se muestra en el Listado 11-48.

```
public interface IEnumerable<out T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
  public IEnumerable<T> Take<T>(int n)
  {
    int k = 0;
    foreach (T x in this)
    {
      if (k < n)
      {
        yield return x;
        k++;
      }
      else yield break;
    }
  }
}
```

Listado 11-48 Hipotética implementación de un método Take dentro de IEnumerable

La clase que implemente `IEnumerable` podría aprovecharse de esta implementación predeterminada o bien dar su propia implementación del método `Take`.

Pero lamentablemente C# no permite colocar ninguna implementación dentro de una interface, de modo que el código del Listado 11-48 no es válido. Sin embargo, mediante los métodos extensores (capítulo 10) se podría hacer la implementación del método `Take` del Listado 11-49

```
public static class Iteradores
{
  //...
  public static IEnumerable<T> Take<T>(this IEnumerable<T> fuente, int n)
  {
    int k = 0;
    foreach (T x in fuente)
    {
      if (k < n)
      {
        yield return x;
        k++;
      }
      else yield break;
    }
  }
}
```

Listado 11-49 Implementación de un método extensor Take

Si `lista` es `IEnumerable`, entonces el tal método `Take` puede continuar invocándose en la forma tradicional `Iteradores.Take(lista, 10);` pero también puede invocarse en la notación punto `lista.Take(10)`.

Cuando se encuentra una invocación como la anterior, el compilador procede de la siguiente manera: primero comprueba si en la definición del tipo con el que está declarada la entidad `lista` hay un tal método `Take`; en caso negativo, busca la existencia de un método extensor `Take` en alguno de los espacios de nombre del proyecto. Como también se dice en el capítulo de Herencia, esto se integra con el Intellisense y mostrar los métodos que se pueden aplicar como si fueran métodos del tipo interface (Figura 11-3)

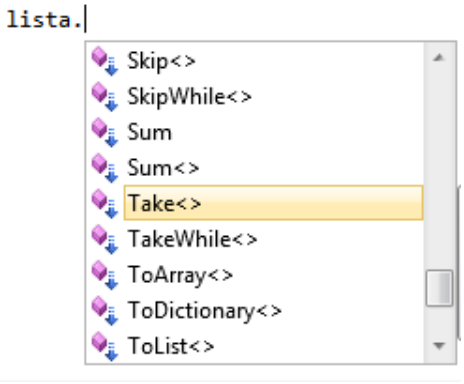

Figura 11-3 Aplicación del Intellisense gracias a los métodos extensores

Hay muchos métodos cuya funcionalidad podría implementarse para `IEnumerable` sin tener que depender de una implementación concreta de `IEnumerable`. C# ofrece la clase estática `Enumerable`, la cual ofrece una amplia variedad de estos métodos extensores. Algunos de ellos se pueden ver en la propia Figura 11-3, por ejemplo el método `IEumerable<T> Skip(IEnumerable<T> source, int count)` devuelve los restantes elementos de `source` después de haber obviado los primeros `count`. Invitamos al lector a implementar un tal método. Antes de ver la documentación de la clase `Enumerable`, trate de concebir algunos métodos que Ud. incluiría en una tal clase. Sobre estos métodos de `Enumerable` se habla también en el Capítulo 12 (Programación Funcional) en el que se presenta el lenguaje de consulta LINQ.

11.10.1 Métodos Extensores y Polimorfismo en el caso de interfaces

Aunque en el capítulo 10 se introdujeron los métodos extensores con la herencia, realmente heredando y definiendo una nueva clase que añada los nuevos métodos siempre podría servir de raíz para una nueva jerarquía. Sin embargo, esto no se puede hacer con las interfaces porque al derivar una interface de otra tampoco se le pueden poner implementaciones a los métodos. De ahí que la mayor utilidad de los extensores está asociada a las interfaces.

Pero realmente los métodos extensores no son métodos de instancia lo que nos puede traer ciertas confusiones en caso de polimorfismo. Si en la implementación de una clase `C` que implemente a una interface `IC` se diese una implementación propia de un método extensor `M` de `IC`, entonces no se aplicaría el polimorfismo cuando se invoque a `M` a través de un objeto del tipo `C` que sea el valor de una entidad declarada de tipo `IC`.

Suponga programado el método extensor `Count` para `IEnumerable<T>` y una clase `MiList` que es `IEnumerable` pero que implementa su propio `Count` (Listado 11-50)

```
static class Utils
{
  //...
  public static int Count<T>(this IEnumerable<T> fuente)
  {
    int count = 0;
    foreach (T x in fuente) count++;
    return count;
  }
}
  class MiList<T> : IEnumerable<T>
  {
    List<T> lista;
    public MiList()
    {
      lista = new List<T>();
    }
    public int Count()
    {
      return lista.Count;
    }
    public void Add(T x)
    {
      lista.Add(x);
    }
    public IEnumerator<T> GetEnumerator()
    {
      return lista.GetEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
      return lista.GetEnumerator();
    }
    //...Otros métodos
  }
//...
```

Listado 11-50 Método extensor Count y polimorfismo

Ejecute el código del Listado 11-51 para que observe la diferencia. Cuando se calcula `granLista.Count()` se aplica el `Count` de `Lista`, que es eficiente porque no tiene que recorrer la lista para saber cuántos elementos tiene; pero cuando se calcula a través de la variable `enteros`, que es de tipo `IEnumerable<int>`, entonces se aplicará el `Count` método extensor de `IEnumerable`, que como no puede conocer cuál es la implementación concreta del `IEnumerable` que tiene como valor la variable `enteros`, entonces lo que hace es recorrer los elementos para poderlos contar.

```
MiLista<int> granLista = new MiLista<int>();
for (int k = 0; k < 10000000; k++) granLista.Add(k);
crono.Restart();
c = granLista.Count();
crono.Stop();
```

```
Console.WriteLine(crono.ElapsedMilliseconds);
IEnumerable<int> enteros = granLista;
crono.Restart();
c = enteros.Count();
crono.Stop();
Console.WriteLine(crono.ElapsedMilliseconds);
```

Listado 11-51 Usando el Count propio y el Count extensor de IEnumerable

Como muestran los resultados del Listado 11-51, esta incongruencia con el polimorfismo pudiera ser realmente un inconveniente cuando una implementación concreta de un `IEnumerable` haga una implementación más eficiente de `Count`.

En general, podría lograrse el efecto del polimorfismo si el método extensor averiguara usando Reflection (tema que se trata en el Capítulo 15 Tipado Dinámico) cuál es el tipo real del objeto al que se aplica el método y si este tiene un método con la misma signatura. Pero ya esto es más engorroso de programar. De todos modos, tenga presente que la utilidad de los extensores es para extender la funcionalidad de tipos que ya existen y fueron definidos como `sealed` o para mantener la sintaxis de la notación de objetos logrando el efecto de añadirle código a una interface `IEnumerable`, como se ha hecho en el caso de algunos de los métodos de `Enumerable`.

11.11 Structs. Definiendo nuestros tipos por valor

La memoria para representar los objetos instancias de un tipo definido por una clase se reserva en un área llamada *heap*. Se dice que las clases constituyen "tipos por referencia", por la forma en que se manejan los objetos almacenados en la memoria. C# introduce el concepto de `struct` como alternativa para definir un tipo en lugar de usar una clase. Los structs[5] se comportan en muchos casos como las clases, siendo su mayor diferencia la forma de almacenamiento en la memoria. Contrariamente a los objetos creados a partir de clases, los structs se consideran "tipos por valor" y por tanto los objetos creados a partir de ellos se almacenan en un área de la memoria denominada pila. Como se menciona en el Capítulo 3, los tipos simples en C# son "tipos por valor". Los structs pueden considerarse entonces como una alternativa a las clases para crear "tipos por valor" personalizados.

Un struct se define de forma similar a una clase, usando la palabra reservada `struct` en lugar de `class`. El Listado 11-52 muestra el código de un struct `Color` para representar colores mediante las componentes RGB[6].

```
struct Color {
      int r, v, a;

      public Color(int rojo, int verde, int azul) {
            r = rojo; v = verde; a = azul;
      }
```

[5] Aunque puede traducirse por estructura, para evitar confusión con otros uso de la palabra estructura en este libro se ha decidido dejar el término en inglés.
[6] RGB, por sus siglas en inglés: Red, Green, Blue.

```
        public int Rojo { get { return r; } set { r = value; } }
        public int Verde { get { return v; } set { v = value; } }
        public int Azul { get { return a; } set { a = value; } }

        public override bool Equals(object obj)
        {
              if (obj is Color) {
                  Color c = (Color)obj;
                  return c.r == r && c.v == v && c.a == a;
              }
              return false;
        }
        public override string ToString() {
              return string.Format("R:{0}, V:{1}, A:{2}", r, v, a);
    }
}
```

Listado 11-52. Struct Color

Debido a la forma en que se almacenan en la memoria los objetos de un tipo `struct`, su uso suele ser en algunos casos más eficiente que al tratar con objetos creados a partir de tipos por referencia. Esto no significa que al diseñar un tipo deba preferirse usar struct en lugar de clase. Generalmente se suelen definir como `struct` aquellos tipos no demasiado complejos, con pocas componentes y una funcionalidad que no necesitará ser redefinida.

Otra diferencia notable entre clases y structs es que los últimos no soportan herencia, por lo que no pueden ser abstractos ni sus métodos o propiedades ser `virtual`. Por tanto no existe polimorfismo ni se aplica el principio de sustitución cuando se trabaja con structs. Este es el costo a pagar por la eficiencia que proporcionan los *structs.* No obstante un *struct* puede implementar todas las interfaces que quiera.

Aunque los structs no soportan herencia, sí forman parte de la jerarquía descendiente de la raíz `Object`, y por tanto heredan sus métodos, que sí pueden por lo tanto ser redefinidos según convenga. En el Listado 11-52Listado 11-52 se muestra una implementación del método `Equals` heredado de la clase `Object`, que devuelve `true` si las componentes de ambos objetos tienen los mismos valores, y `false` en otro caso. Asimismo, se ha redefinido el método `ToString` para devolver también la información de un color como una representación textual indicando los valores de cada componente.

```
Color color1 = new Color(10, 20, 30);
Color color2 = color1;
color1.Rojo = 50;

Console.WriteLine(color1);
Console.WriteLine(color2);
```

Listado 11-53. Usando el struct Color

De acuerdo con la definición del struct `Color`, si se ejecuta el código del Listado 11-53, se escribirá en la ventana de consola

```
R:50, V:20, A:30
```

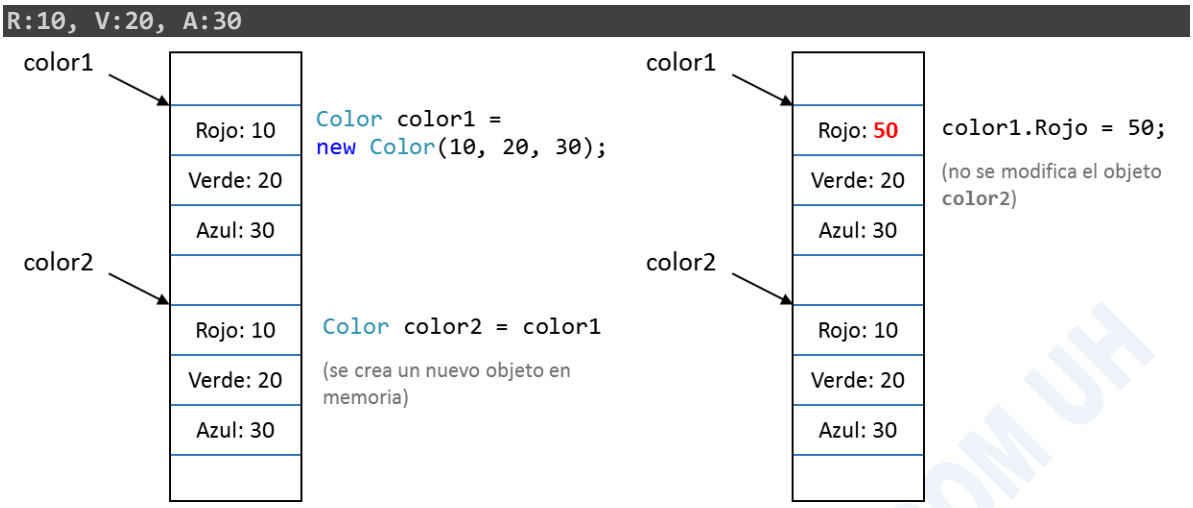

Figura 11-4. Comportamiento de los structs **en memoria**

Note que este comportamiento (Figura 11-4) se debe a que el tipo `Color` se maneja por valor y no por referencia. Es decir, que al hacer `Color color2 = color1`, se crea en la memoria un nuevo objeto independiente de `color1`, con los mismos valores para cada una de sus componentes. Luego, al hacer `color1.Rojo = 50`, no se afecta la componente `Rojo` de `color2`.

Los tipos `Point` y `Color` del espacio de nombres `System.Drawing` y el tipo `DateTime` del espacio de nombres `System`, son structs y por tanto se tratan como tipos "por valor".

11.11.1 Boxing y Unboxing entre referencias y valores

El lenguaje C# define tipos por valor y tipos por referencia para optimizar la ejecución de los programas que los usan. Sin embargo, en algunas situaciones puede necesitarse asignar un tipo por valor a una variable declarada como un tipo por referencia. El ejemplo más general es el de poder asignar "cualquier cosa" a una variable de tipo `Object`.

A una variable de tipo `Object` puede asignársele un objeto de otro tipo definido por una clase, ya sea definida en .NET o definida por terceros. En este caso se pone en práctica el principio de sustitución, y el compilador detecta como válida esta asignación, pues toda clase hereda implícitamente de `Object`. Es por eso que el método `WriteLine` de la clase `Console` puede recibir como parámetro cualquier objeto (incluyendo los de los tipos básicos que caen en la misma categoría de tipo por valor que los structs) sin importar su tipo, pues una de sus sobrecargas recibe `Object`.

```
Fecha fecha = new Fecha(12, 5, 2008);
Console.WriteLine(fecha);
```

En ocasiones puede resultar necesario asignar un objeto de un tipo tratado por valor a una variable de tipo `Object`, para usarlo en algún contexto polimorfo, como el caso del Listado 11-53, donde se pasaron objetos de tipo `Color` al método `WriteLine`. Esto puede hacerse en C# y "por detrás del telón"; lo que ocurre se denomina **boxing**. La acción de

boxing ocurre cuando se asigna un tipo por valor a un tipo por referencia, como es el caso de `Object`. Siendo así, puede hacerse lo siguiente

```
int x = 5;
object obj = x; // Boxing
```

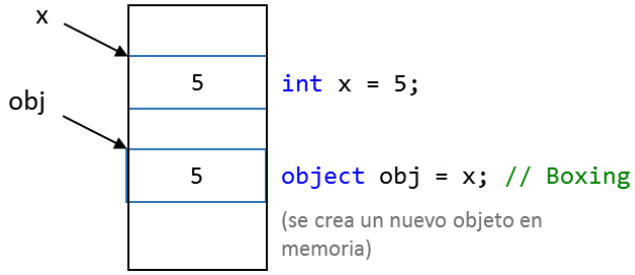

Figura 11-5. Boxing

Como se ilustra en la Figura 11-5, al asignar la variable `x` a la variable `obj`, se crea una copia del objeto en memoria.

La operación contraria se conoce como **unboxing**, e implica convertir un tipo por referencia en un tipo por valor usando el operador de *cast*.

```
x = (int)obj;
```

En el caso de los tipos básicos quizás no resulte notable la utilidad del boxing y unboxing, salvo para la uniformidad del `WriteLine`, pero si trasladamos su aplicación al escenario de los structs quizás resulte más claro.

Los structs son tipos que se tratan por valor, por lo que al asignarlos a una variable de tipo `Object` ocurre el boxing. Los structs pueden implementar interfaces, y cuando un objeto creado a partir de un struct se asigna a una variable cuyo tipo es una interfaz, **¡también ocurre boxing!**

Para ilustrar el boxing-unboxing con structs veamos un ejemplo donde se ha definido una interface `ICredencial` y un struct `Credencial` que la implementa (Listado 11-54).

```
interface ICredencial {
  string Titular { get; }
  DateTime FechaExpiracion { get; }
  void Renovar(int dias);
}
public struct Credencial : ICredencial {
  public Credencial(string titular) {
    this.titular = titular;
    this.fechaExpiracion = DateTime.Today.AddDays(30);
  }
  private string titular;
  public string Titular {
    get { return titular; }
  }
  private DateTime fechaExpiracion;
  public DateTime FechaExpiracion {
```

```
    get { return fechaExpiracion; }
  }
  public void Renovar(int dias) {
    fechaExpiracion = fechaExpiracion.AddDays(dias);
  }
}
```

Listado 11-54. Interface ICredencial y struct Credencial

Suponga la existencia de un método `RenovarCredencial` que recibe como parámetro un `ICredencial` y que invoca el método `Renovar` definido por esta interface.

```
void RenovarCredencial(ICredencial c) { c.Renovar(10); }
```

De acuerdo con esto, se puede hacer entonces

```
Credencial cred = new Credencial("Juan");
RenovarCredencial(cred); // Boxing implícito
Console.WriteLine(cred.FechaExpiracion);
```

¿Qué fecha se escribiría? ¿La fecha original con la que se creó el objeto `cred` o diez días después de la original? Cómo `Credencial` es un `struct`, sus objetos se tratan "por valor" y al pasarlo como parámetro del método `RenovarCredencial` se hace un boxing con una copia del mismo lo que a su vez hace una copia de la referencia al string `Titular` y una copia de la fecha `FechaExpiracion` ya que el tipo `DateTime` es también `struct`. Por tanto, las modificaciones a la fecha dentro del método `RenovarCredencial` afectan a la copia de la fecha realizada como parte del boxing y no a la fecha original. Es decir, que el código anterior escribirá la fecha con la que originalmente se creó el objeto.

En este caso, el boxing está ocurriendo de forma implícita al pasar el parámetro real del tipo `struct Credencial` al parámetro formal definido de tipo `ICredencial` (Figura 11-6).

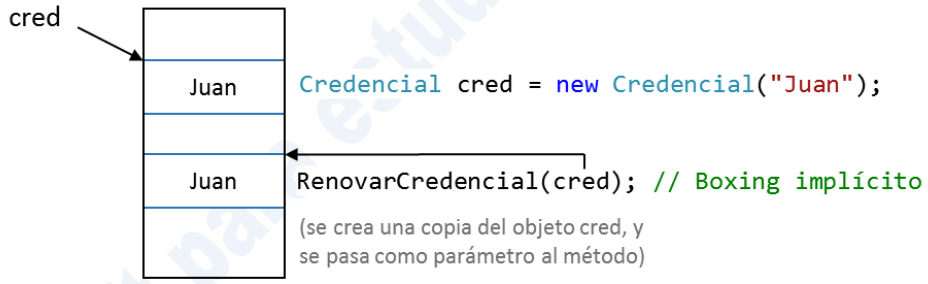

Figura 11-6. Boxing implícito

¿Podría lograrse el efecto de que el método `RenovarCredencial` modifique el objeto original? Sí, y la vía para lograrlo es hacer un uso explícito del concepto de boxing. Con el siguiente código se logra este propósito.

```
ICredencial cred = new Credencial("Juan"); // Boxing explícito
RenovarCredencial(cred);
Console.WriteLine(cred.FechaExpiracion);
```

La sutil diferencia en el texto del código (se cambió el tipo de la variable `cred` de `Credencial` a `ICredencial`) modifica por completo el resultado de ejecutar el método `RenovarCredencial`. En este caso ocurre un boxing cuando de forma explícita asignamos un tipo por valor (el struct `Credencial`) a una variable de tipo por referencia (la variable

`cred` de tipo `ICredencial`) de modo que aunque ha ocurrido un boxing del objeto creado por el `new`, el objeto de tipo `Credencial` al que tenemos acceso es al referido por la variable `cred`. Luego, al método `RenovarCredencial` se le está pasando la referencia al mismo objeto que al que tenemos acceso a través de `cred` y es por eso que la fecha que resulta modificada es la misma (Figura 11-7).

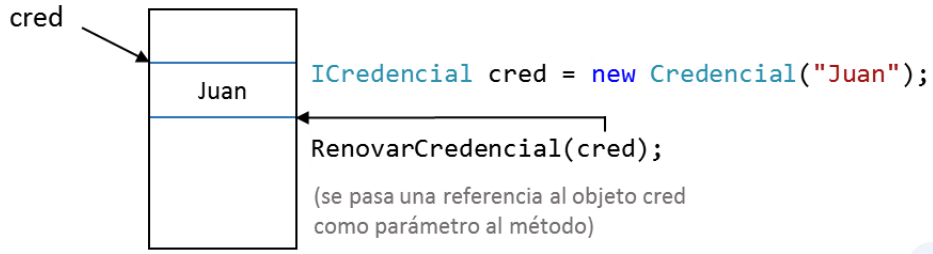

Figura 11-7. Boxing explícito

En resumen, los structs pueden resultar apropiados cuando se trate de tipos relativamente sencillos, con una funcionalidad bien definida, que no necesitará ser extendida o adaptada mediante herencia, y que por razones de eficiencia no convenga modelarse con clases.

En algunas situaciones en las que se requiere interoperar con otras plataformas o lenguajes, puede resultar una exigencia trabajar con structs en lugar de clases. Un ejemplo de ello es cuando se necesita invocar desde C# algunas funciones definidas en bibliotecas externas y en otros lenguajes, como es el caso del API de Windows definido en C++.