Las colecciones

Hemos aprendido a utilizar los arreglos para guardar grupos de información y trabajar más fácilmente. Si bien los arreglos son muy útiles y sencillos de usar, en algunas ocasiones están limitados en su funcionamiento y son poco flexibles para guardar grupos de datos dinámicos. En este capítulo aprenderemos a utilizar las colecciones. Éstas nos brindan muchos de los beneficios de los arreglos y nos dan la flexibilidad necesaria para guardar datos dinámicos.

LAS COLECCIONES MÁS IMPORTANTES

Las colecciones son **estructuras de datos** que nos permiten guardar en su interior cualquier tipo de información. Existen diferentes tipos de colecciones y la forma co- mo se guarda, se accede y se elimina la información en cada una de ellas es distinta. En los arreglos nosotros teníamos que indicar la cantidad de elementos que el arre- glo debía tener. En las colecciones esto no es necesario, ya que es posible agregar elementos **dinámicamente**. Esto quiere decir que cuando el programa se está eje- cutando podemos adicionar o borrar sus elementos.

En otros lenguajes de programación cuando se desea tener este tipo de estructuras generalmente es necesario programarlas antes de poder usarlas. Sin embargo C# nos provee de las colecciones más importantes y podemos utilizarlas directamente sin tener que hacer ningún tipo de desarrollo previo.

Las coleccione que aprenderemos en este capítulo son: **ArrayList, Hashtable, Queue, y Stack**. También aprenderemos un nuevo tipo de ciclo que nos facilita- rá la utilización de estas colecciones.

El ArrayList

La primera colección que aprenderemos se conoce como **ArrayList**, que guarda la información como si fuera una lista. Y sobre esta lista es posible realizar diferentes actividades con los elementos almacenados. Entendemos al **ArrayList** como un arre- glo que puede cambiar su tamaño según lo necesitemos.

Puede guardar cualquier tipo de dato, por lo que lo podemos usar para enteros, cadenas, flotantes o incluso para tipos definidos por nosotros mismos. **ArrayList** es una clase en C#, por lo que va a tener métodos o funciones que nos permiti- rán trabajar con los datos.

El **ArrayList** tiene una propiedad que llamamos **capacidad**, que indica el tamaño que ocupa la lista. También tenemos el **conteo**, el cual nos dice cuántos elementos está guardando en su interior.

Para entender cómo funciona **ArrayList** crearemos una pequeña aplicación y en ella realizaremos las operaciones más importantes.

III EL NAMESPACE DE LAS COLECCIONES

Para poder utilizar las colecciones necesitamos que nuestro programa utilice el namespace de **System.Collections**. Sin este namespace las colecciones no serán reconocidas y tendremos problemas al compilar el programa. Hay que recordar que los namespace a utilizar se colocan al inicio del programa antes de nuestro código.

Declaración de un ArrayList

En nuestro programa podemos tener tantos **ArrayList** como sean necesarios, pero es necesario declararlos primero. La declaración se lleva a cabo de la siguiente manera:

```
ArrayList datos = new ArrayList();
```

Lo primero que necesitamos es indicar **ArrayList**, ya que éste es el nombre de la clase. Luego colocamos el nombre que va a tener, en nuestro caso es datos. Posteriormente pasamos a la instanciación, la cual se lleva a cabo por medio de **new**. En este ejemplo el constructor de la clase no recibe ningún parámetro.

Si bien el **ArrayList** aumenta su tamaño dinámicamente, es posible instanciar el arreglo con algún valor de capacidad inicial. Esto es útil si sabemos inicialmente cuantos elementos puede contener el **ArrayList**. para hacerlo simplemente colocamos la capacidad inicial entre los paréntesis de la siguiente forma:

```
ArrayList datos = new ArrayList(32);
```

Aquí, datos tiene una capacidad inicial de 32, aunque se encuentre vacío.

Adición de información

Nosotros podemos adicionar cualquier tipo de información al **ArrayList**. Hacerlo es muy sencillo y requiere que usemos un método conocido como **Add()**.

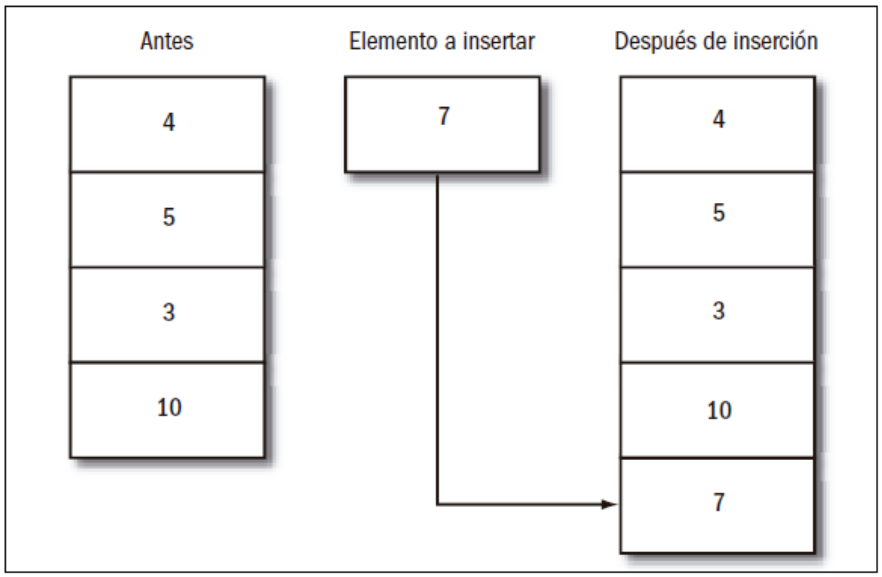

***Figura 1.** Aquí podemos ver cómo el nuevo elemento es colocado al final del ArrayList existente.*

Siempre que se adiciona un elemento al **ArrayList**, este nuevo elemento se agrega al final. Si incorporamos otro elemento, se colocará después del anterior. La adición siempre se lleva a cabo después del último elemento que se encuentre en el **ArrayList**.

El método **Add()** va a necesitar de un único parámetro y este es el dato que queremos guardar. Por ejemplo para guardar un dato de tipo entero podemos hacer lo siguiente

```
datos.Add(7);
```

El dato a guardar también puede ser pasado por medio de una variable:

```
datos.Add(n);
```

Si quisiéramos guardar una cadena, se puede hacer de la siguiente manera:

```
palabras.Add("Hola");
```

El uso de foreach

En la programación orientada a objetos existe un patrón conocido como iterador o iterator, según su sintaxis en el idioma inglés. La principal característica del iterador es permitirnos realizat un recorrido por todos los elementos que existen en una estructura de datos. El recorrido lo hace de forma sequencial, uno por uno. Esto resulta muy útil cuando sabemos que debemos recorrer todos los elementos de la estructura, como un ArrayList, y hacer algo con ellos.

En C# encontramos un iterador en el foreach. Con él es posible que recorramos los elementos, luego ejecutamos alguna acción con ellos y finalizamos cuando la colección ya no tiene más elementos que entregarnos.

La sintaxis es la siguiente:

III LA CAPACIDAD Y EL CONTEO

En varias ocasiones vamos a encontrar que la capacidad del **ArrayList** es mayor que su conteo, pero nunca menor. La capacidad tiende a ser mayor para que las operaciones de inserción de datos sean rápidas. El arreglo guarda los datos hasta que llega a su capacidad y si la cantidad de datos pasa cierto límite, el arreglo crece en su capacidad dejando espacio libre para nuevas inserciones.

```
foreach (tipo identificador in expresión)
     sentencia
```

El **tipo** está relacionado con la clase de información que guarda la colección. Si nuestra colección guarda enteros, entonces el tipo debe de ser **int** y así sucesivamente para cada tipo de dato. Luego tenemos que tener una variable que represente al elemento que se encuentra en la lista, esta variable es el **identificador** y luego por medio de esta variable identificador podremos hacer algo con la información de ese elemento.

La **expresión** simplemente es la colección o el arreglo que vamos a recorrer con el ci- clo. La **sentencia** es el código que se va a repetir para cada vuelta del ciclo. La senten- cia puede ser sencilla o se puede colocar un bloque de código en caso de ser necesario. Como ya conocemos bien los arreglos, veamos cómo podemos utilizar este ciclo para mostrar los contenidos de un arreglo. Supongamos que tenemos un arreglo de enteros que se llama **costo** y al que hemos introducido valores. Para mostrarlo con este ciclo, haremos lo siguiente:

```
foreach(int valor in costo)
{
     Console.WriteLine("El valor es {0}", valor);
}
```

En este caso estamos indicando que el tipo de dato con el que vamos a trabajar es entero. La variable **valor** cambiará su contenido dependiendo del elemento del arreglo que estemos recorriendo en ese momento. Luego simplemente indicamos que el arreglo a recorrer es **costo**. Adentro del ciclo mostramos un mensaje indicando el contenido de **valor** para esa vuelta del ciclo.

Ahora es posible continuar nuestro aprendizaje sobre C#, más específicamente seguiremos revisando las características de las colecciones.

EL ÍNDICE DEL NUEVO ELEMENTO

El método **Add()** regresa un valor de tipo entero. Este valor indica el índice donde fue guardado un nuevo elemento. Muchas veces ignoramos este valor, pero en algunos casos puede resultar- nos útil. Hay que recordar que el índice en el cual se encuentra un elemento puede variar, ya que es posible eliminar elementos del **ArrayList** también.

Cómo acceder a la información de un ArrayList

La colección **ArrayList** nos permite acceder a sus elementos por medio de un índice, algunas colecciones no lo permiten, pero **ArrayList** sí. Esto es bueno, pues podemos trabajarla de forma similar a un arreglo, pero con la flexibilidad de ser dinámica.

Si tenemos un **ArrayList** llamado datos y deseamos imprimir el elemento **2**, lo podemos hacer de la siguiente manera:

```
Console.WriteLine("El dato es {0}",datos[2]);
```

De igual forma podemos utilizar el valor del elemento en una expresión:

```
impuesto = datos[2] * 0.15f;
```

Y podemos también asignar un valor determinado:

```
datos[2] = 5;
```

Cómo obtener la cantidad de elementos en un ArrayList

En muchas ocasiones es útil saber cuántos elementos tenemos en el **ArrayList**. A diferencia del arreglo tradicional, éste puede cambiar su tamaño durante la ejecución de la aplicación y conocer el tamaño nos evita problemas con los valores de los índices al acceder el arreglo.

Afortunadamente es muy sencillo hacerlo, simplemente tenemos que leer el valor de la propiedad **count** del **ArrayList**. Esta propiedad es un valor entero.

```
elementos = datos.Count;
```

En este caso la variable elementos tendrá la cantidad de elementos en el arreglo datos.

No debemos olvidar que si elementos tiene el valor de 5, los índices irán de 0 a 4.

Insertar elementos

Hemos visto que podemos adicionar elementos y que éstos son colocados al final del **ArrayList**, sin embargo también es posible llevar a cabo una inserción. La inserción permite colocar un elemento nuevo en cualquier posición válida del arreglo.

Para lograr esto usamos el método **Insert()**. Este método necesita de dos parámetros, el primer parámetro es el índice donde deseamos insertar el elemento y el segundo parámetro es el elemento a insertar. La figura que se muestra a continuación, se encarga de mostrar cómo trabaja la inserción.

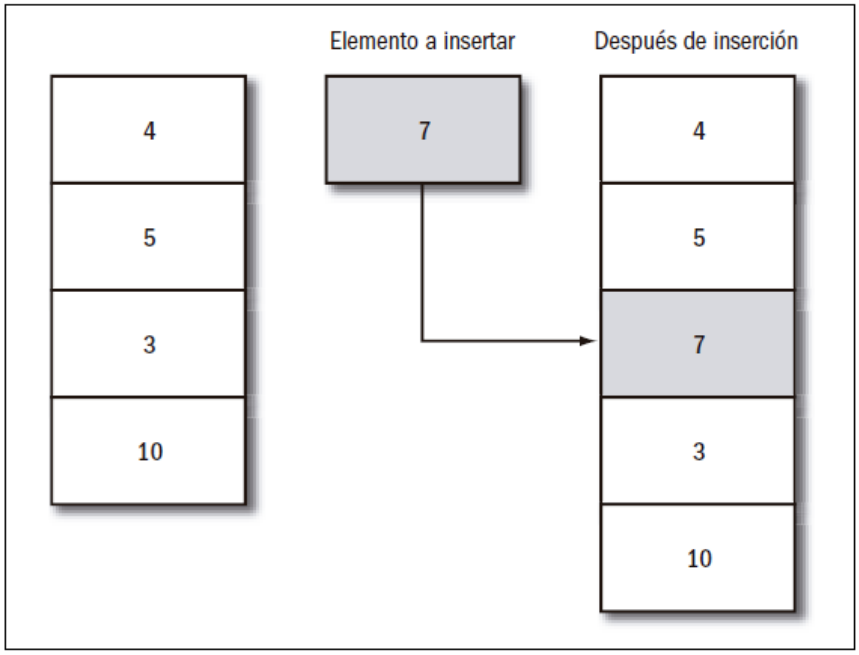

***Figura 2.** Podemos observar cómo se lleva a cabo la inserción del elemento, debemos notar cómo algunos elementos cambian de índice.*

Por ejemplo, si deseamos insertar el valor de 5 en el índice 2, hacemos lo siguiente:

```
datos.Insert(2, 5);
```

El **ArrayList** crecerá su tamaño según sea necesario.

LA CAPACIDAD DEL ARRAYLIST

Si en alguna de nuestras aplicaciones necesitáramos saber cuál es la capacidad del **ArrayList**, es posible hacerlo al leer el valor de propiedad **Capacity**. Esta propiedad es de tipo entero. Si lo que deseamos es reducir la capacidad del **ArrayList**, se puede usar el método **TrimToSize()**, pero debemos tener cuidado para no perder información útil.

Para eliminar un elemento

Es posible eliminar cualquier elemento del **ArrayList** y hacerlo de forma muy sencilla. Lo único que necesitamos es conocer el índice del elemento a eliminar. El índice es un valor entero y debe ser válido. El método que se encarga de llevar a cabo la eliminación es **RemoveAt()**.

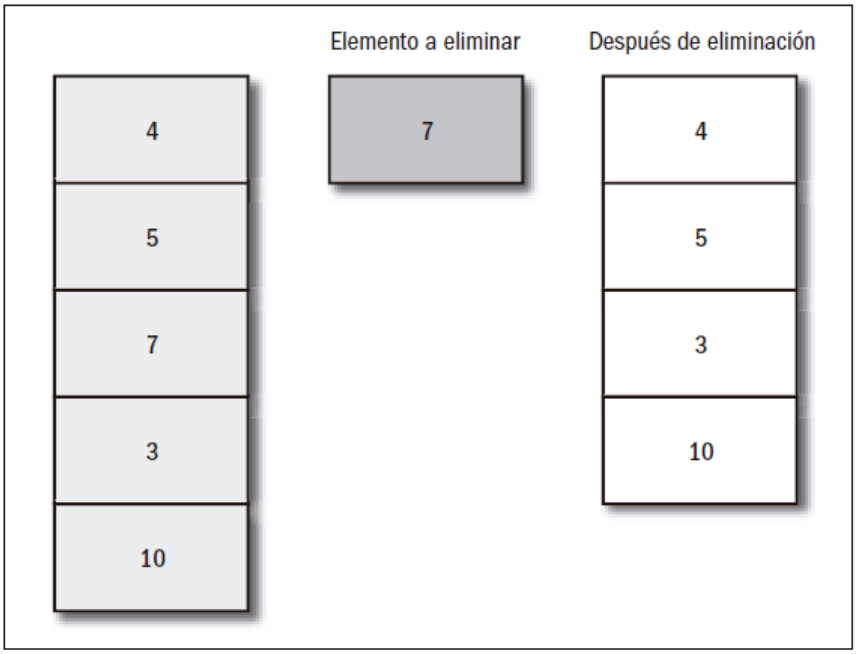

***Figura 3.** Al eliminar el elemento, desaparece del ArrayList. Los demás elementos se reorganizan y sus índices pueden modificarse.*

Este método solamente necesita de un parámetro, que es el índice del objeto que deseamos eliminar. Por ejemplo si queremos eliminar el elemento que se encuentra en el índice 7, podemos hacer lo siguiente:

```
datos.RemoveAt(7);
```

Para encontrar un elemento

Con los **ArrayList** es posible saber si un elemento en particular se encuentra adentro de él. Para lograr esto hacemos uso del método **IndexOf()**. Este método requiere de un solo parámetro que es el objeto a buscar adentro del **ArrayList**. El método nos regresa un valor entero.

Este valor es el índice donde se encuentra la primera ocurrencia del elemento, esto es debido a que podemos tener el elemento guardado en diferentes posiciones. Si el elemento no se encuentra en el **ArrayList**, entonces simplemente recibimos el valor de **-1**.

Si lo que deseamos es buscar el índice donde se encuentra el elemento **7** en nuestro **ArrayList**, hacemos lo siguiente:

```
índice = datos.IndexOf(7);
```

Ahora la variable índice tiene la locación donde se encuentra el elemento **7**.

Ejemplo con ArrayList

Ahora podemos construir un ejemplo que use el **ArrayList** y los diferentes métodos que es posible usar con él. Empecemos por declarar nuestras variables y colocar lo necesario para crear el **ArrayList** y adicionar unos datos inicialmente.

```
static void Main(string[] args)
            {
                    // Variables necesarias
                    int indice=0;
                    int cantidad=0;

                    // Declaramos el ArrayList
                    ArrayList datos = new ArrayList();


                    // Adicionamos valores al ArrayList
                    datos.Add(7);
                    datos.Add(5);
                    datos.Add(1);

                    Console.WriteLine("Tenemos inicialmente los
                           datos:");
                    Imprime(datos);

}
```

Para que nuestra aplicación muestre el arreglo crearemos una pequeña función llamada Imprime. Esta función recibe el **ArrayList** a imprimir y hace uso del **foreach** para mostrar cada elemento del **ArrayList**.

```
static void Imprime(ArrayList arreglo)
{

        foreach (int n in arreglo)

                Console.Write(" {0},",n);

        Console.WriteLine("\n----------");
}
```

Si ejecutamos la aplicación obtendremos lo siguiente:

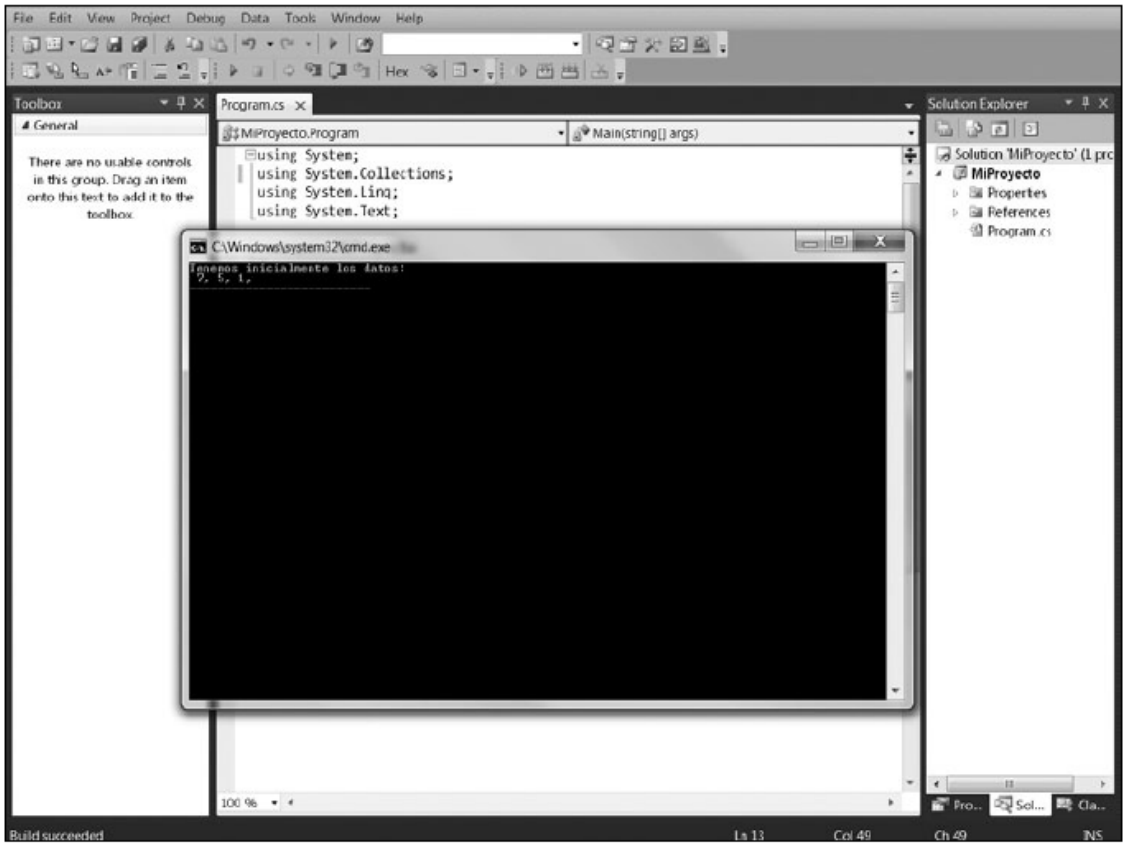

***Figura 4.** Podemos observar cómo se han mostrado los elementos del ArrayList.*

Vamos a experimentar un poco y haremos crecer al arreglo insertando tres nuevos valores. Para el último valor obtendremos el índice donde fue colocado.

```
// Hacemos crecer el ArrayList
datos.Add(4);
datos.Add(5);

// Obtenemos el indice
indice=datos.Add(10);

Console.WriteLine("Despues de hacerlo crecer:");
```

```
Imprime(datos);
Console.WriteLine("El ultimo elemento tiene el
       indice {0}",indice);
Console.WriteLine("\n----------");
```

Ejecutemos el programa y veremos el siguiente resultado.

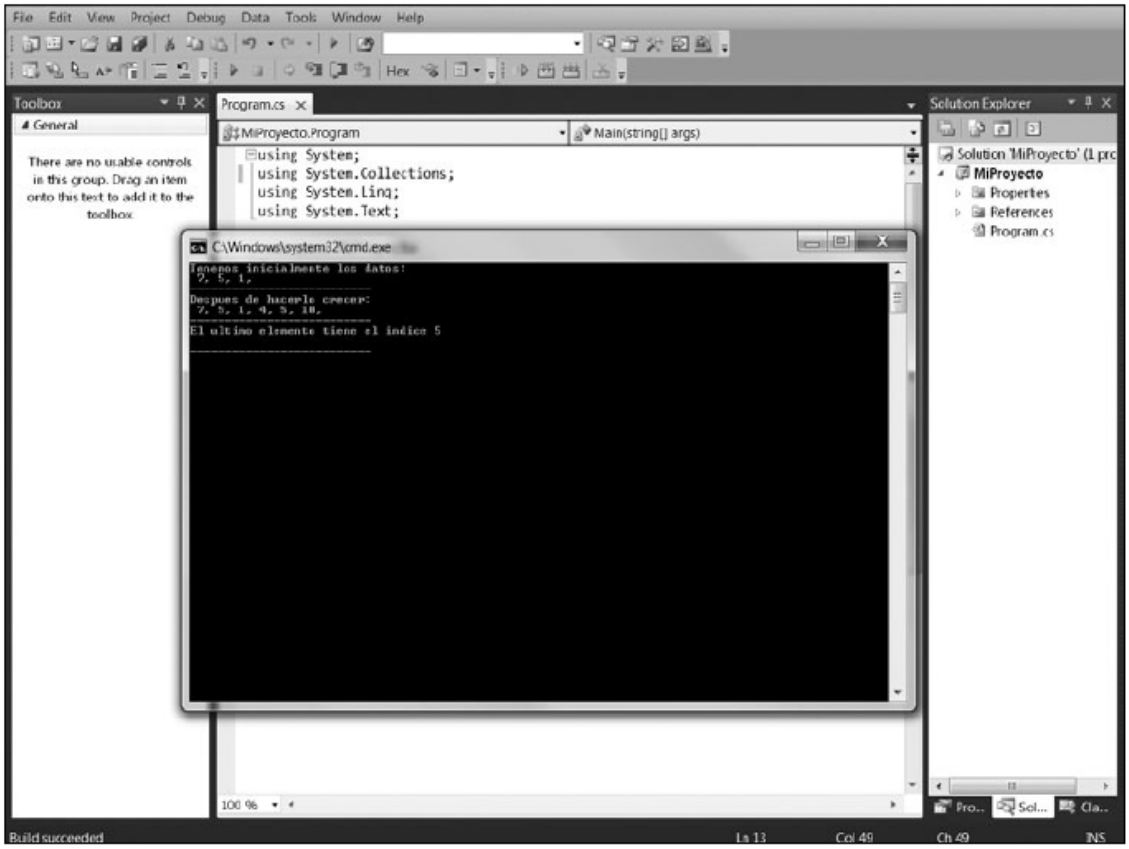

***Figura 5.** Podemos verificar que el ArrayList ha crecido y vemos cómo hemos obtenido el índice de la última inserción.*

Ahora utilizaremos el índice para leer el valor de un elemento y modificar el valor de otro. En este caso la sintaxis es muy similar a la de los arreglos.

```
// Imprimimos un elemento en particular
Console.WriteLine("El valor en el indice 2 es
        {0}",datos[2]);
Console.WriteLine("\n----------");

// Modificamos un dato
datos[3]=55;

Console.WriteLine("Despues de la modificacion:");
Imprime(datos);
```

Si compilamos este ejemplo, podremos observar cómo se llevó a cabo el cambio del elemento adentro del arreglo correspondiente.

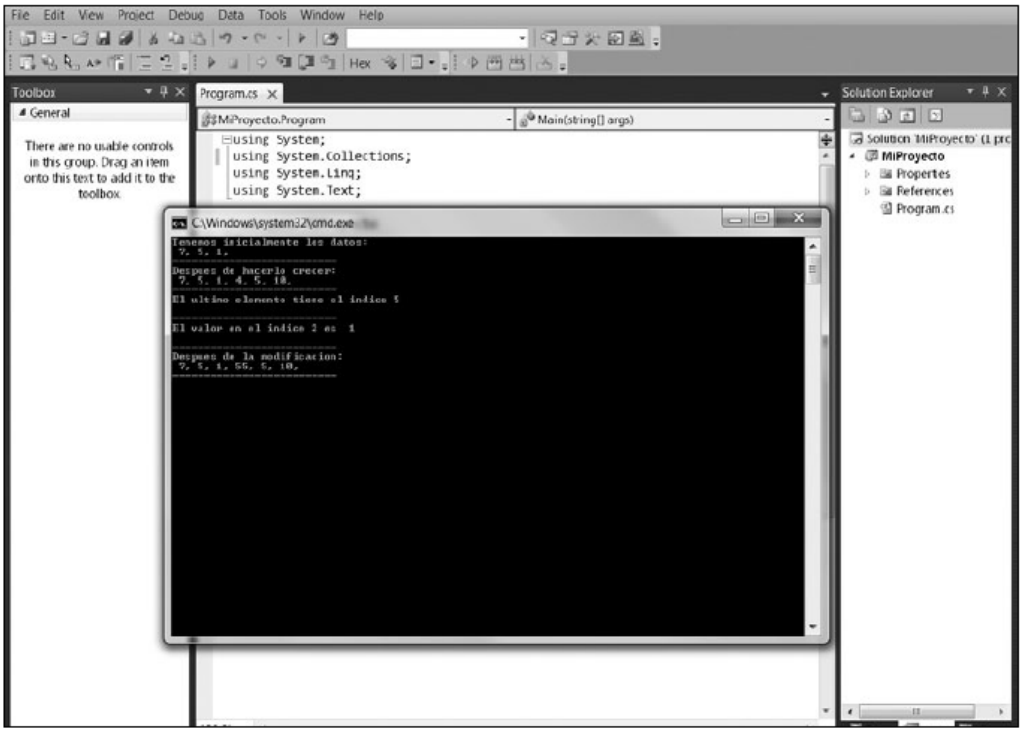

Figura 6. *Hemos obtenido el contenido de un elemento y modificado otro.*

En este momento obtendremos la cantidad de elementos que contiene el arreglo y la desplegaremos en la pantalla.

III PROBLEMAS CON LOS KEY DE LAS HASHTABLE

Cuando hacemos uso de las **Hashtable**, cada elemento que coloquemos debe de tener un **key** único. El **key** no se debe repetir, si lo hacemos podemos experimentar problemas con nuestro programa. La razón de esto es que el **key** es usado para encontrar la posición en el **Hashtable** del elemento y una repetición crearía conflicto al tener dos elementos diferentes en la misma posición.

```
// Obtenemos la cantidad

cantidad=datos.Count;
Console.WriteLine("La cantidad de elementos es
        {0}",cantidad);
Console.WriteLine("\n----------");
```

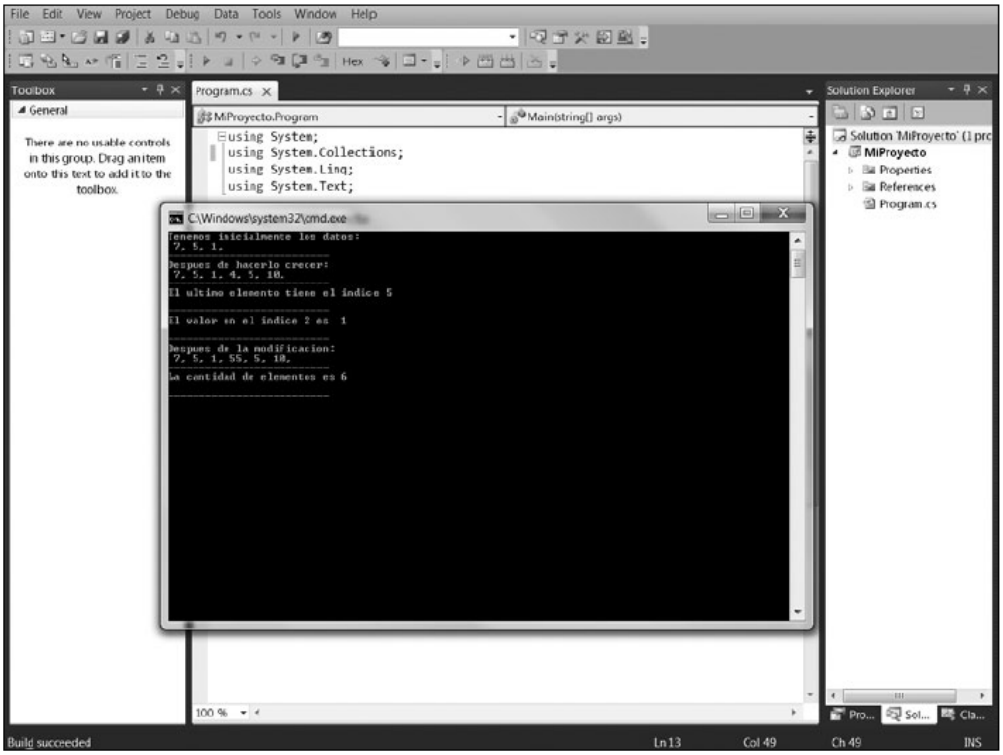

Figura 7. *Hemos obtenido la cantidad de elementos en el* ***ArrayList****.*

Si deseamos insertar un elemento en alguna posición en particular es posible hacerlo.

```
// Insertamos un elemento
datos.Insert(2,88);

Console.WriteLine("Despues de la insercion:");
Imprime(datos);
```

CUIDADO CON EL RANGO

Al igual que con los arreglos tradicionales, a **ArrayList** le debemos tener cuidado cuando lo accedamos por medio de un índice ya que si intentamos acceder a un elemento con un índice inexistente obtendremos un error al ejecutar el programa. No debemos usar índices negativos ni mayores a la cantidad de elementos que se tienen. Los índices también están basados en cero.

Al ejecutar el programa podemos ver que fue insertado en el lugar de adicionado.

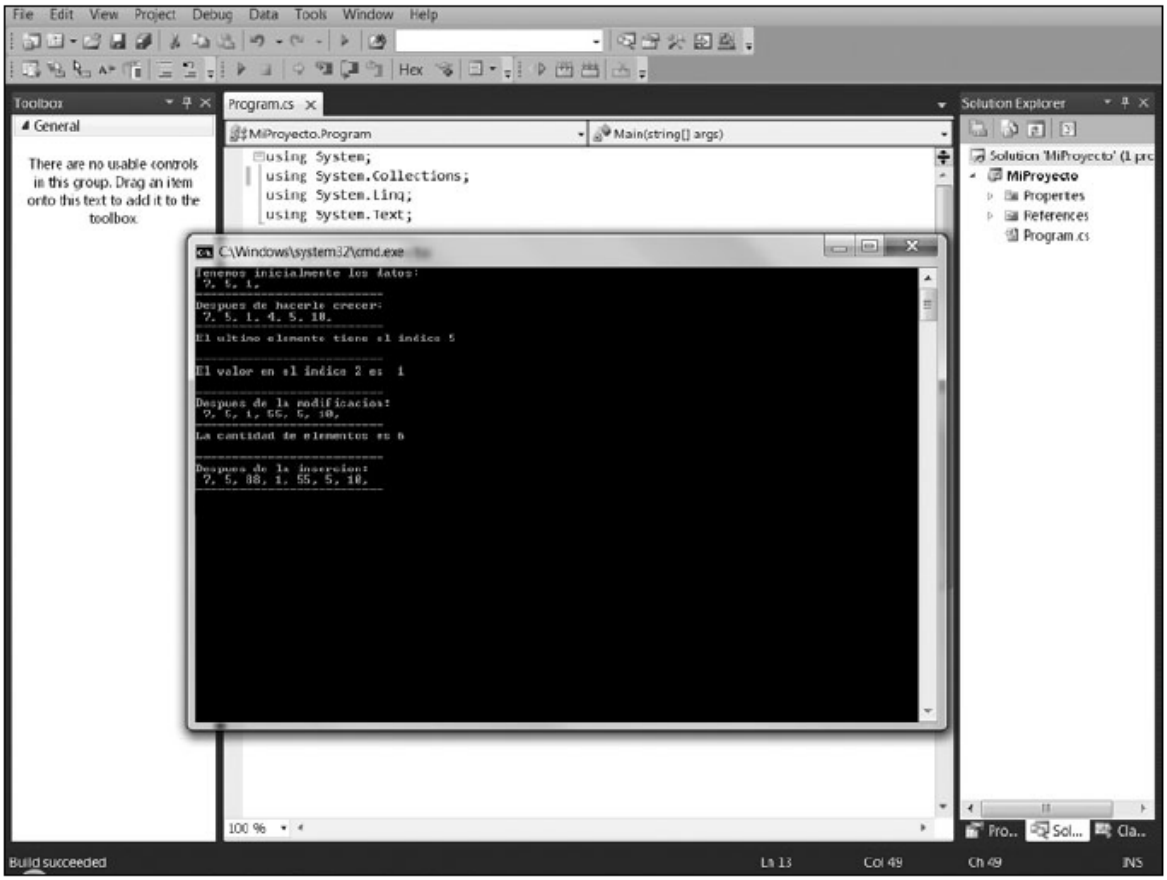

***Figura 8.** La inserción puede llevarse a cabo en cualquier lugar del ArrayList, la adición solo al final.*

Si lo que necesitamos es eliminar un elemento lo hacemos de la siguiente manera:

```
// Eliminamos un elemento

datos.RemoveAt(4);
Console.WriteLine("Despues de la eliminacion:");
Imprime(datos);
```

III OTRA FORMA DE TOMAR EL ELEMENTO

El método **Pop()** tomó el elemento que se encuentra en la parte superior del **Stack** y lo regresa al exterior, pero el elemento ya no pertenece al **Stack**. Hay otro método conocido como **Peek()** que lee el valor del elemento en la parte superior del **Stack** y lo regresa al exterior, pero no elimina el elemento del **Stack**.

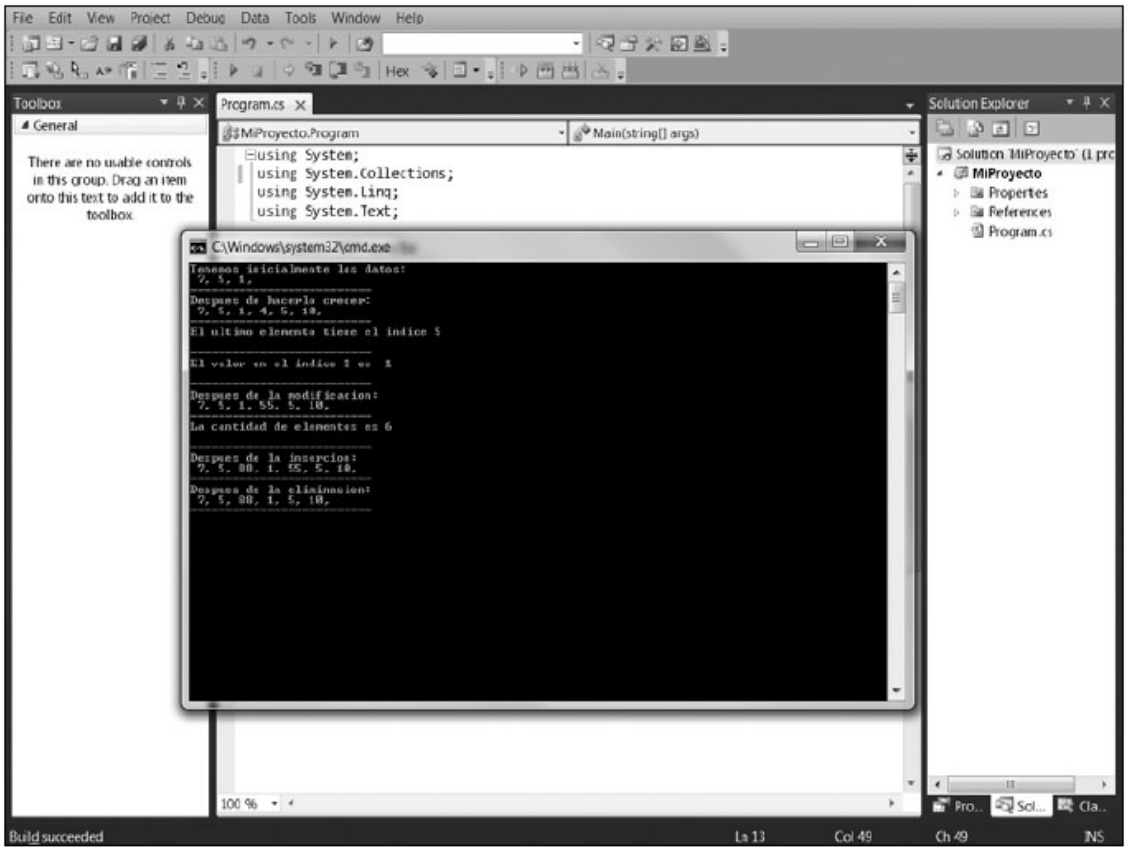

Figura 9. *El elemento ha desaparecido del ArrayList.*

Lo último que nos falta en la generación de nuestra aplicación, es dotarla de la capacidad de encontrar el índice donde se encuentra un elemento en particular. Para ello debemos agregar el código que se muestra a continuación.

```
// Encontramos el indice donde se encuentra el
       primer 5
indice=datos.IndexOf(5);
Console.WriteLine("Elprimer 5 se encuentra en
       {0}",indice);
Console.WriteLine("\n----------");
```

Luego de agregar el trozo de código propuesto en el bloque anterior, nuestra aplicación será capaz de encontrar el índice; y como ya se halla completa lucirá similar al ejemplo que podemos ver en la imagen mostrada a continuación:

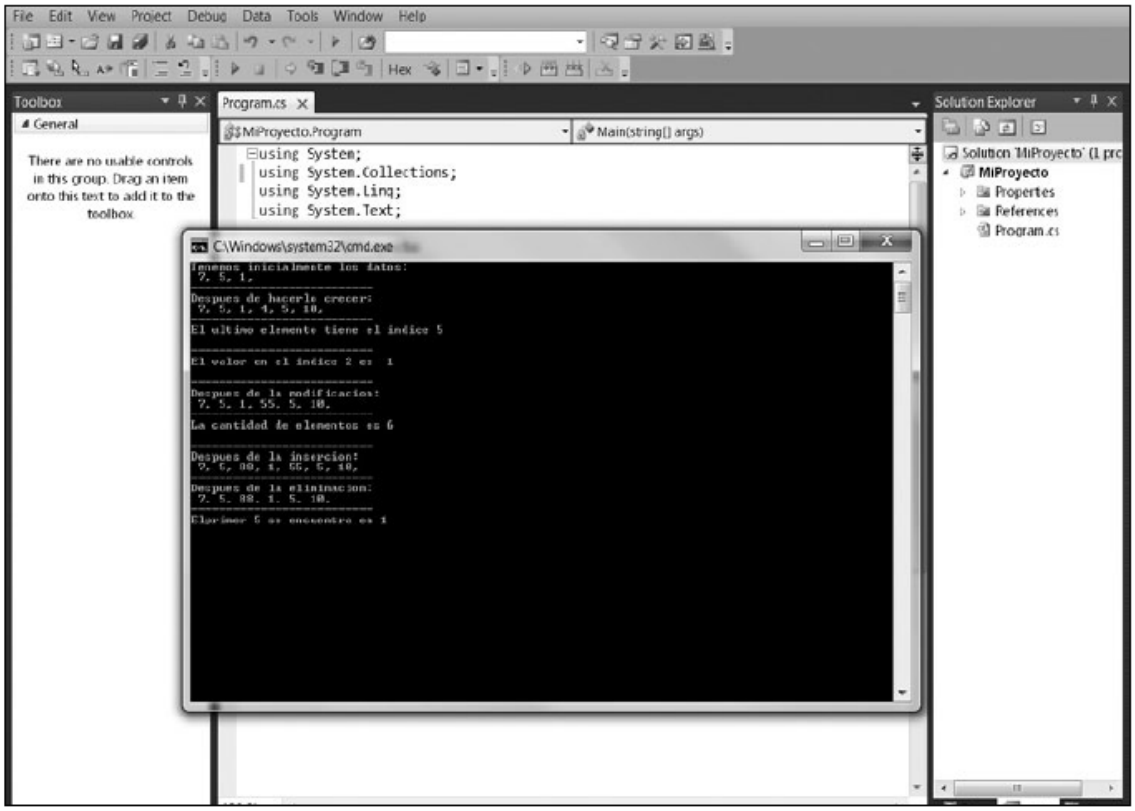

Figura 10. *La primera ocurrencia del valor de 5 es en el índice 1.*

El Stack

Ahora empezaremos a conocer otro tipo de colección. A esta colección se la conoce como **Stack** o **pila**, nos permite guardar elementos y cambia su tamaño de forma dinámica, sin embargo trabaja en forma diferente al arreglo y al **ArrayList**.

El **Stack** es una estructura de tipo **LIFO**. LIFO es el acrónimo en inglés para *Lastin-first-out*, es decir el primero que entra, el último que sale. Para entender su funcionamiento podemos imaginar una pila de platos. El primer plato que colocamos queda hasta la base, el siguiente se colocará encima y así sucesivamente. Como el primer plato queda hasta abajo, no lo podemos sacar directamente pues la pila se derrumbaría. Al sacar los platos, debemos tomar el que se encuentre hasta arriba de la pila primero y así continuar.

III LA CAPACIDAD DEL STACK

El **Stack** al igual que el **ArrayList** tiene una capacidad y ésta crece dinámicamente. Cuando instanciamos el **Stack** adquiere la capacidad de default. Si necesitamos crear un **Stack** con determinada capacidad, la forma de hacerlo es colocando el valor de capacidad entre los paréntesis del constructor al momento de instanciarlo.

El efecto de colocar nuevos elementos en la parte superior del **Stack** se conoce como **Push**. Esto se muestra en la siguiente figura:

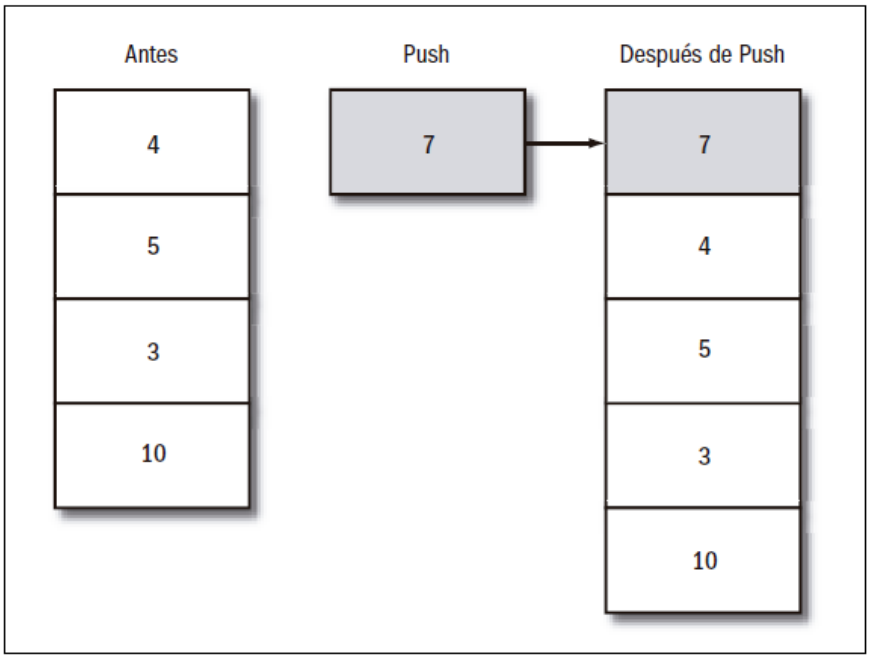

***Figura 11**. Podemos observar cómo **Push** inserta nuevos elementos en la parte superior.*

Cuando tomamos un elemento de la parte superior del **Stack** se conoce como **Pop**. En la figura podemos observar cómo esto sucede.

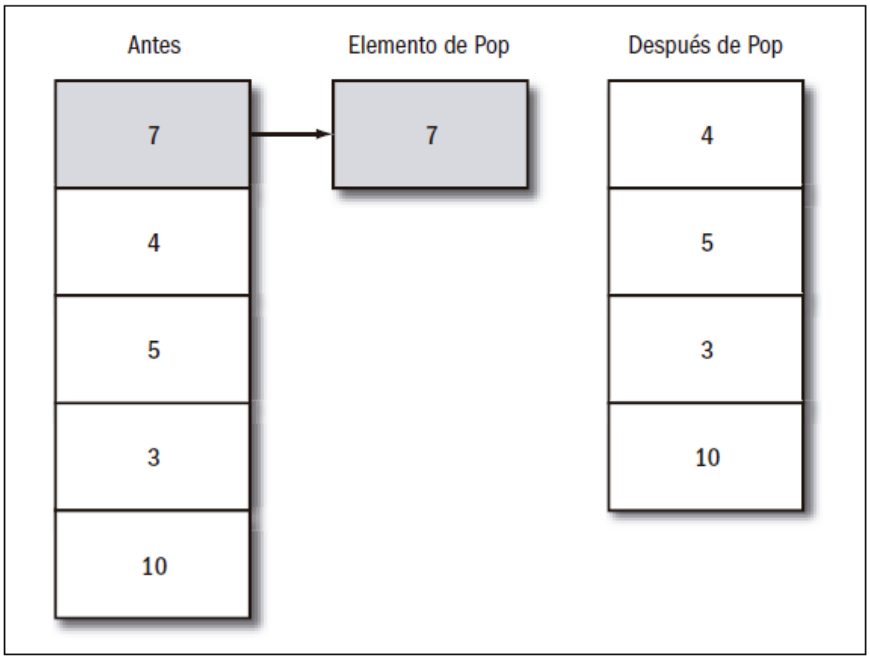

***Figura 12. Pop** toma el elemento que se encuentra en la parte superior del Stack.*

Debido a este comportamiento con **Push** y **Pop** podemos entender cómo el primer objeto que se coloca adentro del Stack es el último en poder salir.

Como crear el Stack

Como cualquier otra colección el **Stack** necesita ser instanciado para poderlo utilizar.

C# nos provee de la clase **Stack** y adentro de esta clase encontramos todos los métodos necesarios para trabajar con él. La instanciación simplemente será la creación de un objeto de esta clase. Si deseamos crear un **Stack** hacemos lo siguiente:

```
Stack miPila = new Stack();
```

Hemos creado un **Stack** llamado **miPila**. Nosotros podemos usar cualquier nombre que sea válido. Una vez instanciado podemos empezar a colocar información en él.

Cómo introducir información al Stack

Para introducir información al **Stack** usamos el método **Push()**. Este método coloca el nuevo elemento en la parte superior del **Stack**. El método necesita únicamente de un parámetro que es el elemento que deseamos insertar. Podemos utilizar el método **Push()** cuantas veces sea necesario para colocar la información.

```
miPila.Push(7);
miPila.Push(11);
miPila.Push(8);
```

En este ejemplo se introduce primero el elemento **7** y luego sobre él se coloca el elemento **11**. En seguida se coloca un nuevo elemento en la parte superior, que en este caso es **8**. El Stack resultante se muestra en la siguiente figura:

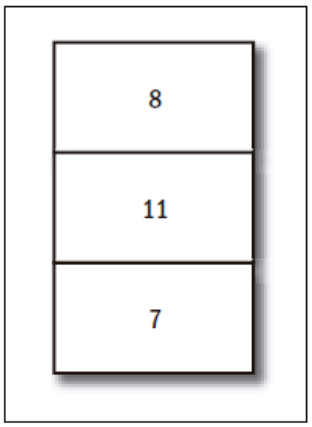

***Figura 13.** El Stack resultante nos muestra como el elemento 7 ha quedado en la base.*

Cómo obtener información del Stack

Si lo que necesitamos es obtener la información que está contenida en el **Stack** lo podemos hacer al tomar el elemento que se encuentra en la parte superior del **Stack**.

Para lograr esto hacemos uso del método **Pop()**. Este método no necesita ningún parámetro y regresa el elemento correspondiente.

Por ejemplo, si deseamos tomar el elemento de nuestro **Stack**:

```
int valor = 0;
…
valor = (int)miPila.Pop();
```

Siguiendo el ejemplo anterior la variable valor ahora tendrá en su interior **8**. Esto lo vemos en la siguiente figura:

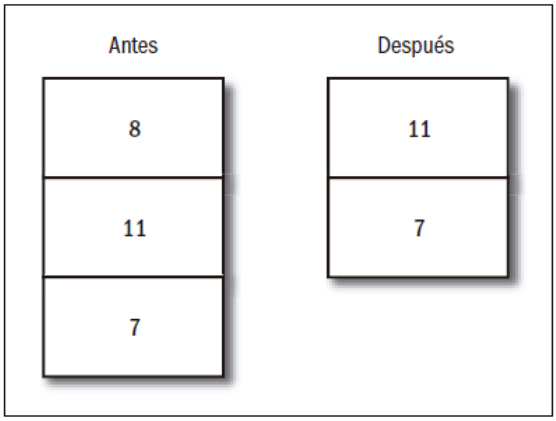

Figura 14. *Se ha tomado el elemento de la parte superior del Stack. Hay que notar que el elemento no continúa siendo parte de él.*

Uso de foreach para recorrer el Stack

Si necesitamos recorrer el **Stack**, es posible hacerlo por medio del uso de **foreach**. El uso de esta alternativa es similar al del **ArrayList**.

```
foreach( int n in miPila)
     Console.WriteLine("{0}",n);
```

Podemos usar el **foreach**, generalmente recorreremos el **Stack** por medio de **Pop()**.

Para obtener la cantidad de elementos del Stack

Es posible conocer la cantidad de elementos que tiene el **Stack** y hacerlo es muy sencillo.

Debemos leer la propiedad **Count** del **Stack**. Esta propiedad nos regresa un valor entero con la cantidad de elementos. El uso es equivalente al del **ArrayList**.

```
cantidad = miPila.Count;
```

III EL CASTING

Existe una clase a partir de la cual descienden todas las demás clases en C#, esta clase se conoce como **Object**. Muchos métodos regresan objetos de tipo **Object**, por lo que es necesario indicar el tipo correcto bajo el cual debemos de usarlo. Para hacer esto hacemos uso del casting. Por ejemplo para hacer un casting a entero hacemos lo siguiente: Valor = (int)fila.Dequeue();

Para limpiar los contenidos del Stack

Si deseamos eliminar todos los elementos del **Stack** de forma rápida lo podemos hacer al usar el método **Clear()**. Este método no necesita de ningún parámetro y solamente debe ser usado cuando sepamos que debemos borrar los elementos del **Stack**.

```
miPila.Clear();
```

Para saber si el Stack tiene un elemento

Si deseamos saber si adentro del **Stack** se encuentra un elemento en particular, podemos hacer uso del método **Contains()**. Este método necesita de un parámetro que es el objeto a encontrar adentro del **Stack**. El método regresa un valor de tipo **bool**.

Si el objeto se encuentra el valor es **true**, pero si no se encuentra es **false**.

```
bool enStack = false;
…
enStack = miPila.Contains(7);
```

Creación de una aplicación

Podemos hacer un programa que muestre como usar los métodos del **Stack**. Para esto crearemos una aplicación que presente las operaciones del **Stack**. El contenido se mostrará para ver los cambios que se tienen debidos al **Push** y al **Pop**.

```
        static void Main(string[] args)
        {
```

```
            // Variables necesarias

            int opcion=0;
            string valor="";
            int numero=0;
            bool encontrado=false;

            // Creamos el stack

            Stack miPila= new Stack();
```

```
do
{
        // Mostramos menu
        Console.WriteLine("1- Push");
        Console.WriteLine("2- Pop");
        Console.WriteLine("3- Clear");
        Console.WriteLine("4- Contains");
        Console.WriteLine("5- Salir");
        Console.WriteLine("Dame tu opcion");
        valor=Console.ReadLine();
        opcion=Convert.ToInt32(valor);

        if(opcion==1)
        {
                // Pedimos el valor a introducir
                Console.WriteLine("Dame el valor a
                        introducir");
                valor=Console.ReadLine();
                numero=Convert.ToInt32(valor);
```

```
                // Adicionamos el valor en el stack
                miPila.Push(numero);
        }

        if(opcion==2)
        {

                // Obtnemos el elemento
                numero=(int)miPila.Pop();
```

```
                // Mostramos el elemento
                Console.WriteLine("El valor obtenido
                        es: {0}",numero);
        }

        if(opcion==3)
        {
                // Limpiamos todos los contenidos
                        del stack
                miPila.Clear();
        }
```

```
                        if(opcion==4)
                        {
                                // Pedimos el valor a encontrar
                                Console.WriteLine("Dame el valor a
                                        encontrar");
                                valor=Console.ReadLine();
                                numero=Convert.ToInt32(valor);

                                // Vemos si el elemento esta
                                encontrado=miPila.Contains(numero);

                                // Mostramos el resultado
                                Console.WriteLine("Encontrado -
                                        {0}",encontrado);
                        }

                        // Mostramos la informacion del stack
                        Console.WriteLine("El stack tiene {0}
                                elementos",miPila.Count);
                        foreach(int n in miPila)
                                Console.Write(" {0},", n);
```

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


                }while(opcion!=5);
        }
```

```
    }
}
```

El programa es muy sencillo, en primer lugar declaramos las variables necesarias. Una variable es para la opción seleccionada por el usuario. Otra variable llamada **numero** será usada para los casos en los que tengamos que obtener o colocar un valor numérico en el **Stack**. La variable llamada **encontrado** es de tipo **bool** y tendrá el valor de **true** o **false** dependiendo si el elemento a buscar se encuentra o no en el **Stack**. Como siempre tenemos una variable de trabajo de tipo cadena que nos ayuda a obtener el valor dado por el usuario. Luego simplemente creamos un **Stack** llamado **miPila**.

Tenemos un ciclo **do** que estará repitiéndose hasta que el usuario decida finalizar el programa. En el ciclo lo primero que hacemos es mostrar el menú con las posibles acciones que podemos hacer con el **Stack**. Luego de acuerdo a cada opción manipulamos el **Stack**. Después de la opción hacemos uso del **foreach** para mostrar loscontenidos del **Stack** y poder verificar los cambios realizados. También aprovechamos para mostrar la cantidad de elementos que existen

en el **Stack**. Compilemos y ejecutemos la aplicación. Lo primero que haremos es insertar algunos valores en el **Stack**, para esto seleccionaremos la opción **1**. Coloquemos los valores **5**, **7**, **3**.

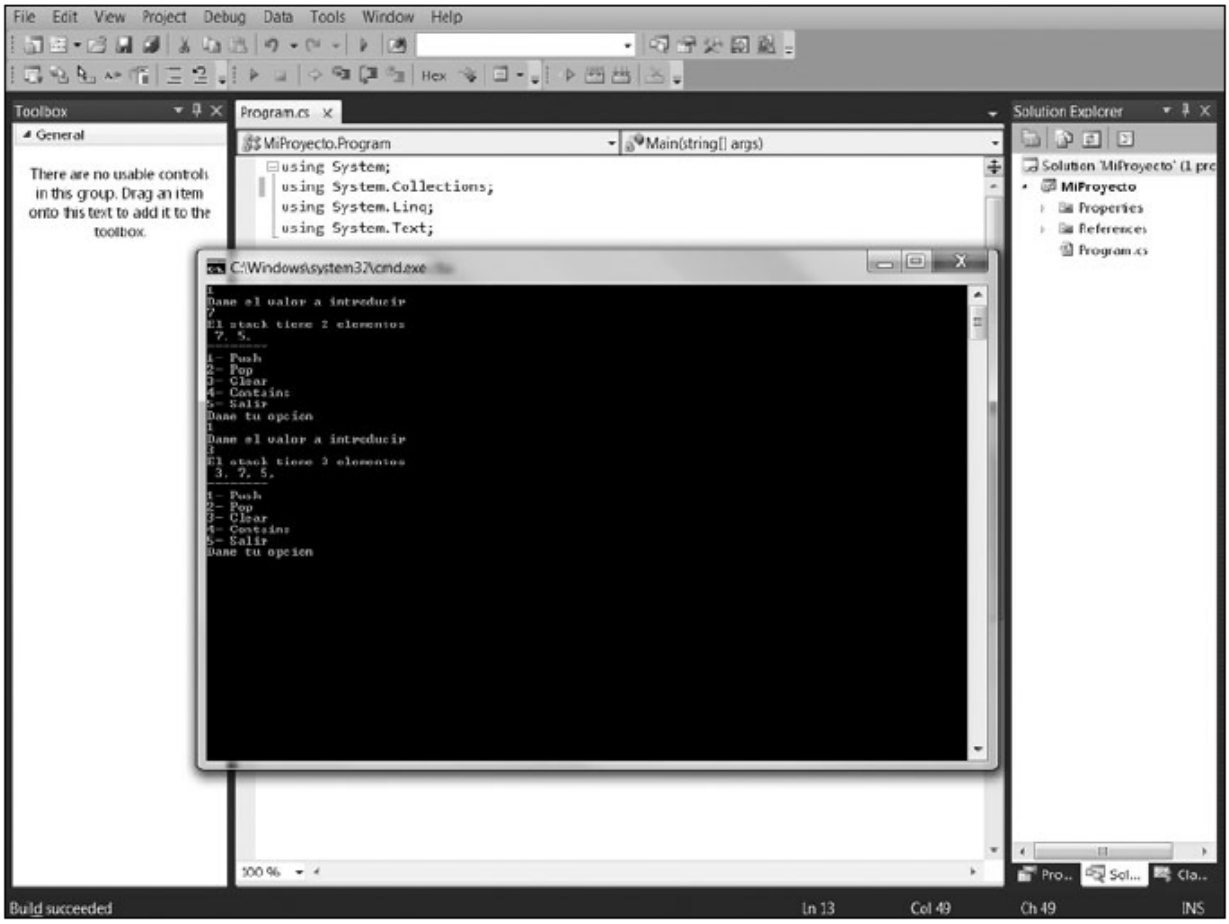

***Figura 15.** Podemos observar cómo el último valor introducido se encuentra al inicio del Stack, el primer valor se encuentra al final.*

Si en este momento hacemos un **Pop**, el **Stack** debe de regresarnos el elemento que se encuentra en la parte superior del **Stack**, en nuestro ejemplo debe ser **3**.

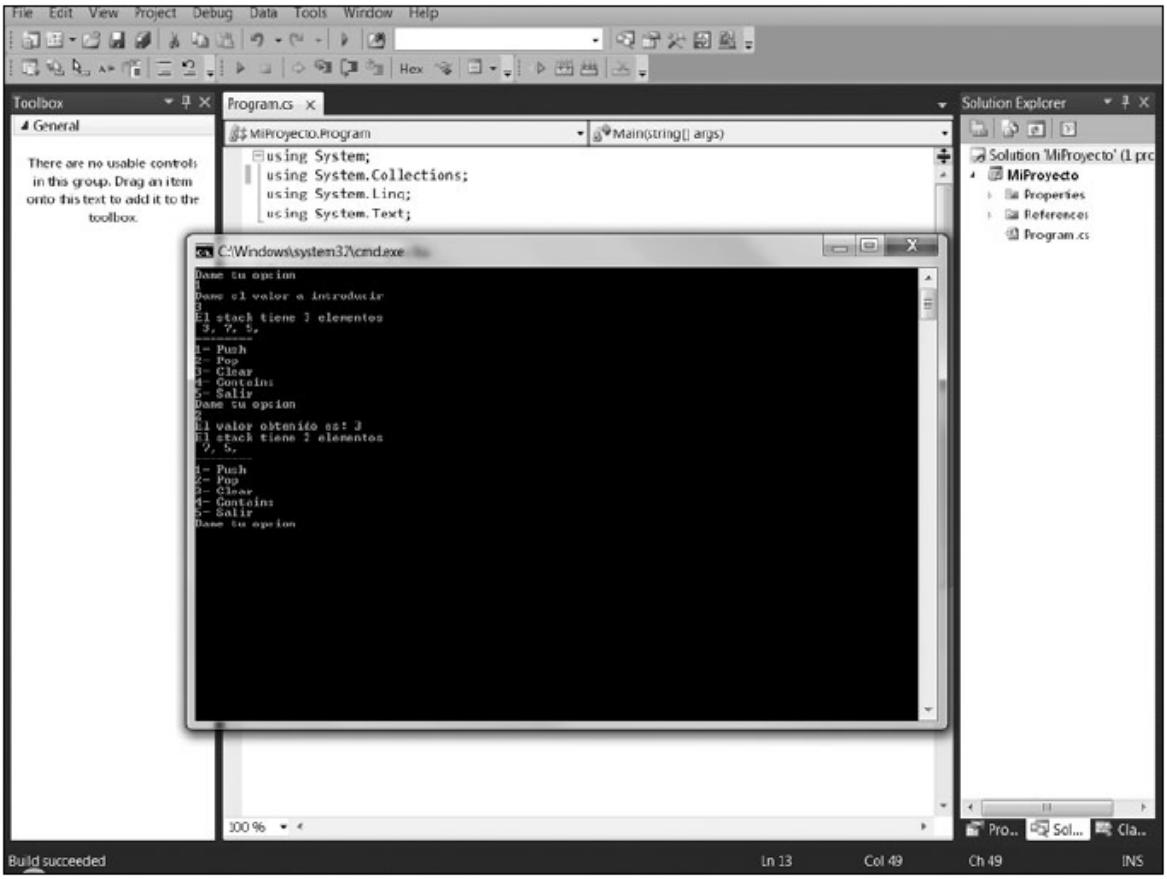

***Figura 16.** Vemos que el valor obtenido es 3 y el Stack ahora solamente tiene dos elementos.*

Ahora podemos ver si el **Stack** tiene un elemento en particular. Para realizar esta operación, nos encargaremos de seleccionar la opción número **4** y para continuar tendremos que preguntar por el elemento número **9**.

Si ahora preguntamos por el elemento **5** obtendremos una respuesta diferente y la variable **encontrado** deberá tener el valor **true**. Lo último que nos queda por hacer es limpiar el **Stack** y para esto seleccionamos la opción **3**.

III EL CONSTRUCTOR DEL QUEUE

Nosotros podemos utilizar el constructor de default del **Queue** el cual no necesita de ningún parámetro, sin embargo, es buena idea consultar MSDN para conocer otros constructores. Por ejemplo tenemos una versión que nos permite colocar la capacidad inicial del **Queue** y otros que nos pueden resultar útiles en nuestras aplicaciones.

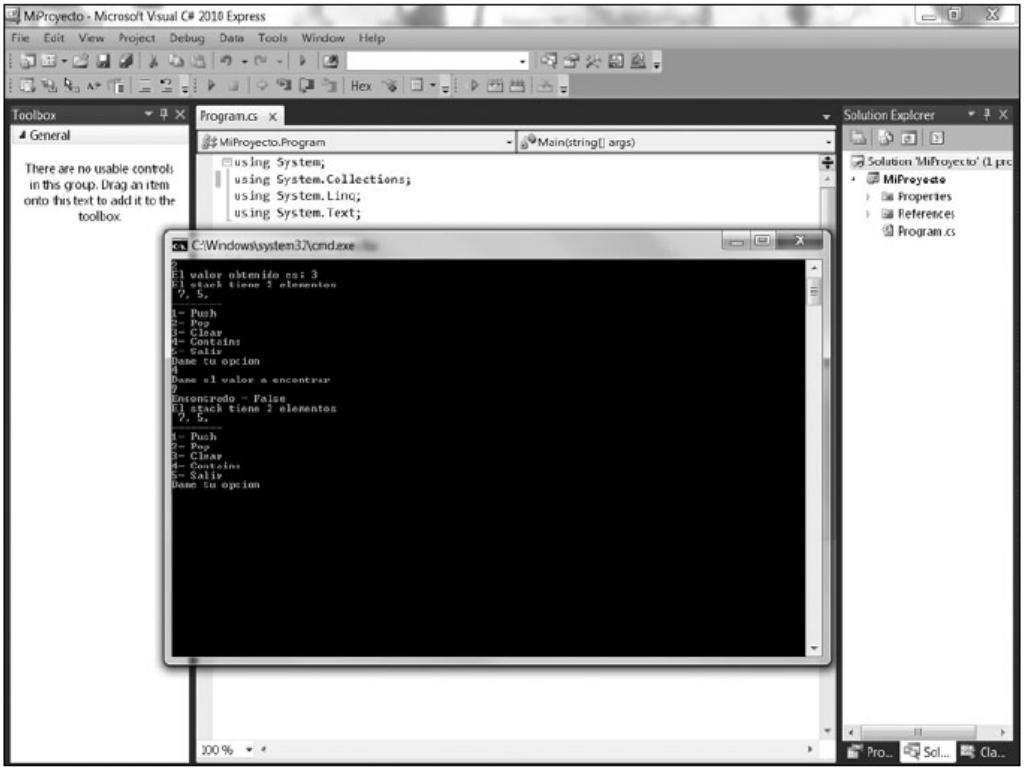

Figura 17. *El Stack nos regresa el valor de false en la variable encontrado pues el elemento no se halla en el Stack.*

El Queue

La siguiente colección que aprenderemos se conoce como **Queue**, algunas veces tam- bién denominada **cola**. Al igual que en el **Stack**, nosotros ya no necesitamos progra- marla, pues C# nos provee una clase con toda la funcionalidad necesaria para poder utilizarla. La clase que debemos de utilizar para poder tener una cola es **Queue**.

La forma como trabaja es diferente al **Stack**, ya que el **Queue** es una estructura de ti- po **FIFO**. Su comportamiento es diferente al **Stack**. Para entender el **Queue**, pode- mos imaginar una fila para comprar los boletos o las entradas al cine. La primera persona en llegar se coloca junto a la taquilla y conforme van llegando otras perso-nas se colocan atrás de ella y así sucesivamente. Sin embargo, en cuanto se habré la taquilla, la primera persona en pasar es la primera que se encuentra en la fila.

III EL COMPORTAMIENTO DE FIFO

Cuando nosotros trabajamos con una estructura de datos que funcione bajo el esquema de **FIFO** debemos tomar en cuenta su comportamiento. **FIFO** es el acrónimo para **First In First Out**, que en español significa: Primero en Entrar, Primero en Salir. En estas estructuras el primer elemen- to que colocamos, es el primer elemento que podemos extraer.

Esto nos indica que cuando insertamos elementos en el **Queue**, éstos se colocan atrás del último elemento insertado o en la parte posterior de éste. Al momento de leer elementos, se toma el que se encuentre en la parte superior del **Queue**. Esto lo podemos ver más fácilmente en la siguiente figura:

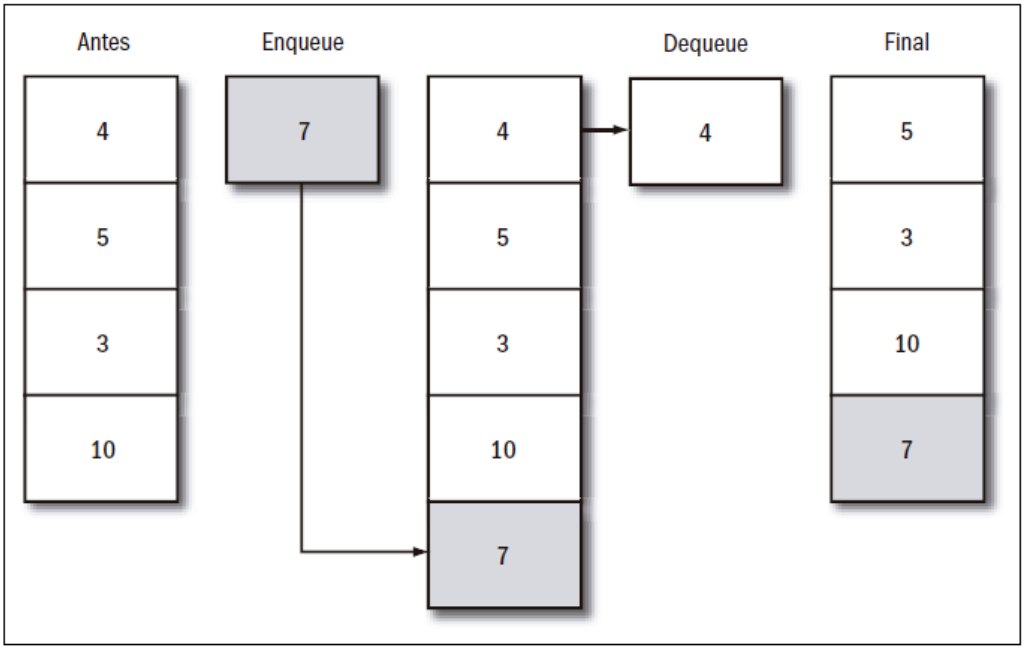

***Figura 18.** Aquí podemos observar el proceso de adición de un elemento y también qué es lo que sucede cuando se extrae un elemento.*

El proceso por medio del cual nosotros insertamos un elemento nuevo al **Queue** se conoce como **Enqueue** y no debemos olvidar que lo hace al final. El acto de extraer un elemento del **Queue** se llama **Dequeue** y siempre toma el elemento que se encuentra en la parte superior.

Declaración del Queue

Para utilizar el **Queue**, lo primero que debemos hacer es crear una instancia de la clase **Queue**. El nombre de la instancia puede ser cualquier nombre válido de variable en C#. Si deseamos crear un **Queue** llamado fila, haremos lo siguiente:

```
Queue fila = new Queue();
```

Una vez instanciado ya es posible hacer uso de fila y también de los métodos que nos provee la clase correspondiente para trabajar con él.

Adición de elementos al Queue

Nosotros podemos adicionar elementos al **Queue** en cualquier momento que lo necesitemos.

El tamaño del **Queue** se modificará dinámicamente por lo que no debemos de preocuparnos por él. Siempre que se adiciona un elemento, este elemento se coloca al final o en la parte baja del **Queue**.

Para poder hacer esto debemos utilizar el método **Enqueue()**, el cual pertenece a la clase **Queue** de C#. Este método es muy sencillo, ya que solamente requiere de un parámetro. En el parámetro colocamos el elemento que deseamos añadir, este método no regresa ningún valor.

Por ejemplo, si deseamos añadir el elemento 7 a nuestro **Queue** llamado fila, será necesario que escribamos el código que se muestra a continuación:

```
fila.Enqueue(7);
```

Otra altrenativa posible es colocar variables en el parámetro adecuado, si realizamos esto, una copia del valor se ubicará en el **Queue**.

Cómo extraer un elemento del Queue

Al igual que podemos colocar un elemento, también es posible extraerlo. Sin embargo, la extracción se lleva a cabo de acuerdo a las reglas del **Queue**. Cuando extraemos un elemento, el elemento que recibimos es el que se encuentra en la parte superior o al inicio del **Queue**. Debemos recordar que no es posible la extracción de los elementos que se encuentren en otras posiciones.

Para llevar a cabo la extracción tenemos que usar un método de la clase **Queue** que se llama **Dequeue()**. El método no necesita de ningún parámetro y regresa el elemento correspondiente. Es importante tener una variable que reciba al elemento o una expresión que haga uso de él.

Si lo que deseamos es extraer un elemento del **Queue**, podremos agregar el código que se muestra en el siguiente bloque:

```
int valor = 0;
…
…
valor = fila.Dequeue();
```

Para observar un elemento del Queue

Cuando hacemos uso del método **Dequeue()**, el elemento es extraído y deja de encontrarse adentro del **Queue** después de esa operación. Sin embargo, podemos observar el elemento. En este caso recibimos el valor del elemento, pero éste no es eliminado del **Queue** después de ser leído.

Para llevar a cabo la observación hacemos uso del método **Peek()**. Este método no necesita de ningún parámetro y regresa el elemento observado. Al igual que con **Dequeue()** debe de existir una variable o una expresión que reciba este valor.

PÉRDIDA DE PRECISIÓN

Algunos tipos de datos, en especial los numéricos, pueden llegar a ser compatibles entre sí. Pero debemos tener algo en cuenta. Cuando convertimos de un tipo mayor a uno menor, por ejemplo de **double** a **float**, es posible que tengamos perdida de información. Esto se debe a que el tipo menor no puede guardar tantos dígitos como el mayor.

Un ejemplo de su uso puede ser:

```
int valor =  0;
…
…
valor = fila.Peek();
```

Para saber si el Queue tiene determinado elemento

Algunas veces podemos necesitar saber si en el interior del **Queue** se guarda un elemento en particular. Esto es posible hacerlo gracias a un método conocido como **Contains()**. Para usar este método, necesitamos pasar como parámetro el elemento que queremos determinar. El método regresará un valor de tipo **bool**. Si el elemento existe adentro del **Queue**, el valor regresado será **true**. En el caso de que el elemento no exista en el interior del **Queue** el valor regresado será **false**. Necesitamos tener una variable o una expresión que reciba el valor del método **Contains()**.

Por ejemplo, si deseamos saber si el elemento 7 existe, podemos hacer lo siguiente:

```
if(miFila.Contains(7) == true)
     Console.WriteLine("El elemento si existe");
```

Para borrar los contenidos del Queue

Si necesitamos eliminar todos los contenidos del **Queue**, es sencillo de hacer por medio del método **Clear()**. Este método ya es conocido por nosotros de colecciones que previamente hemos estudiado. Su utilización es muy sencilla ya que no necesita de ningún parámetro. Solamente debemos tener cuidado, ya que si lo utilizamos en un momento inadecuado, perderemos información que todavía puede ser útil.

Para eliminar todos los elementos del **Queue**, colocamos lo siguiente:

```
miFila.Clear();
```

Para conocer la cantidad de elementos que tiene el Queue

Es conveniente saber la cantidad de elementos que contiene el **Queue**, especialmente antes de llevar a cabo algunas operaciones sobre éste. Cuando deseamos saber la cantidad de elementos existentes, debemos utilizar la propiedad de **Count**. Esta propiedad corresponde a un valor de tipo entero.

III EL CONSTRUCTOR DE LA CLASE

Las clases tienen un método especial llamado **constructor**. Este método es invocado automáticamente en el momento que un objeto de esa clase se instancia. Los constructores pueden o no recibir parámetros y siempre se nombran igual que la clase, se usan principalmente para inicializar correctamente al objeto.

La podemos utilizarla de la siguiente manera:

```
int cantidad = 0;
…
…
cantidad = miFila.Count;
```

Si queremos aprovecharla para hacer una operación en particular y evitar algún error con el **Queue**, puede ser algo como lo siguiente:

```
if(miFila.Count>0)
     valor = (int)miFila.Dequeue();
```

Para recorrer los elementos del Queue

Si necesitamos recorrer los elementos del **Queue**, por ejemplo, para mostrarlos o imprimirlos lo que nos conviene usar es un ciclo **foreach**. Su utilización es muy sencilla y similar a lo que hemos usado con otras colecciones.

```
foreach( int n in miFila)
     Console.WriteLine("{0}",n);
```

Ahora podemos ver un ejemplo que haga uso del **Queue**. En este ejemplo llevaremos a cabo las operaciones más comunes sobre éste. Mostraremos al usuario un menú y dependiendo de la opción seleccionada será la operación a realizar.

```
using System;
using System.Collections;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {

                    // Variables necesarias

                    int opcion=0;
                    string valor="";
                    int numero=0;
                    bool encontrado=false;
```

```
// Creamos el Queue

Queue  miFila= new Queue();
```

```
do
{
        // Mostramos menu
        Console.WriteLine("1- Enqueue");
        Console.WriteLine("2- Dequeue");
        Console.WriteLine("3- Clear");
        Console.WriteLine("4- Contains");
        Console.WriteLine("5- Salir");
        Console.WriteLine("Dame tu opcion");

        valor=Console.ReadLine();
        opcion=Convert.ToInt32(valor);

        if(opcion==1)
        {
                // Pedimos el valor a introducir

                Console.WriteLine("Dame el valor a
                        introducir");
                valor=Console.ReadLine();
                numero=Convert.ToInt32(valor);
```

```
                // Adicionamos el valor en el queue
  miFila.Enqueue(numero);

        }

        if(opcion==2)
        {
                // Obtnemos el elemento
                numero=(int)miFila.Dequeue();
```

OTRO ESTILO DE COLECCIONES

Existe otro estilo de colecciones que son funcionalmente equivalentes a las colecciones que hemos visto, pero que proveen mejor desempeño y más seguridad en el manejo de los tipos de los elementos. Estas colecciones utilizan tipos genéricos para los elementos y se encuentran en el namespace **System.Collections.Generics**

```
                        // Mostramos el elemento
                        Console.WriteLine("El valor obtenido
                                es: {0}",numero);
                }

                if(opcion==3)
                {
                        // Limpiamos todos los contenidos
                                del Queue
                        miFila.Clear();
                }

                if(opcion==4)
                {
                        // Pedimos el valor a encontrar
                        Console.WriteLine("Dame el valor a

                                encontrar");
                        valor=Console.ReadLine();
                        numero=Convert.ToInt32(valor);
```

```
                        // Vemos si el elemento esta
                        encontrado=miFila.Contains(numero);

                        // Mostramos el resultado
                        Console.WriteLine("Encontrado -
                        {0}",encontrado);
                }

                // Mostramos la informacion del stack
                Console.WriteLine("El Queue tiene {0}
                        elementos",miFila.Count);
                foreach(int n in miFila)
                        Console.Write(" {0},", n);

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

        }while(opcion!=5);
```

```
            }

        }
    }
```

El Hashtable

La última colección que aprenderemos en este capítulo se conoce como **Hashtable**.

Es una estructura de datos un poco compleja de implementar, pero afortunadamente C# nos provee de una clase que nos da toda la funcionalidad necesaria y podemos utilizarla tan fácilmente como las colecciones anteriores.

El **Hashtable** es una colección indexada. Es decir que vamos a tener un índice y un valor referenciado a ese índice. Sin embargo, la indexación no se lleva a cabo como en el arreglo o el **ArrayList**. El lugar adentro del **Hashtable** donde se coloca el elemento va a depender de un valor conocido como **key** o **llave**. El valor contenido en **key** es usado para calcular la posición del elemento en el **Hashtable**. El elemento que vamos a colocar se conoce como **value**.

En el **Hashtable** siempre utilizaremos una pareja de **key** y **value** para trabajar con él.

La forma como trabaja el **Hashtable** puede parecer extraña, pero en realidad es una estructura muy eficiente al momento de leer la información. El comprender el funcionamiento interno del **Hashtable** pertenece a un libro de nivel más avanzado, por lo que no tocaremos este tema aquí.

La clase se llama **Hashtable** y pertenece al igual que las demás colecciones al namespace **System.Collections**.

Declaración del Hashtable

La declaración del **Hashtable** es muy sencilla y similar a la declaración de las otras colecciones. Para hacerlo, simplemente creamos una instancia de la clase **Hashtable** y después podemos hacer uso de las operaciones de la colección por medio de los métodos que nos provee.

Si lo que deseamos es declarar e instanciar un **Hashtable** llamado **miTabla**, es posible realizarlo prosiguiendo de la siguiente manera:

```
Hashtable miTabla = new Hashtable();
```

El nombre del **Hashtable** puede ser cualquier nombre válido de variable para C#. Ahora ya podemos aprender a utilizarlo.

OTRAS COLECCIONES

Tenemos otras colecciones que no veremos en el capítulo pero resultan interesantes y útiles de aprender. El conocer cómo trabajan otras colecciones puede ahorrarnos mucho código y tiempo ya que posiblemente existe lo que estamos tratando de hacer. Para poder conocerlas, como siempre, es recomendable visitar MSDN. Algunas de estas colecciones son: **BitArray**, **Dictionary** y **SortedList**.

Adición de elementos al Hashtable

Nosotros podemos adicionar cualquier cantidad de elementos al **Hashtable**. Hacerlo es tan sencillo como con las otras colecciones, pero debemos tomar en cuenta la forma cómo el **Hashtable** indexa su información.

Para insertar un elemento usamos el método **Add()**. Este método a diferencia de los de las otras colecciones, va a necesitar de dos parámetros. En el primer parámetro colocamos el **key** que será usado para indexar al elemento. Este **key** puede ser de cualquier tipo, pero generalmente se utilizan cadenas. El segundo parámetro será el valor o elemento a insertar, también puede ser de cualquier tipo.

Hay que recordar que cuando hagamos uso de esta colección siempre trabajamos la información en parejas **key-value**.

Por ejemplo, podemos usar el **Hashtable** para guardar un producto con su costo. El nombre del producto puede ser el **key** y su costo será el **value**. Si deseamos insertar algunos elementos, quedará de la siguiente forma:

```
miTabla.Add("Pan", 5.77);
miTabla.Add("Soda", 10.85);
miTabla.Add("Atun", 15.50);
```

El lugar donde quedaron colocados estos elementos adentro del **Hashtable** depende del algoritmo de **hashing** que se use. Realmente a nosotros no nos interesa, ya que cualquier acceso que necesitemos lo haremos por medio de **key**.

Recorriendo el Hashtable

Para poder recorrer el **Hashtable**, haremos uso del ciclo **foreach**. Si queremos obtener la pareja **key-value**, nos apoyaremos en una clase conocida como **DictionaryEntry**. El diccionario también guarda parejas de datos.

Por ejemplo, si deseamos proseguir con la impresión de los contenidos del Hashtable, podemos colocar el código que se muestra a continuación:

```
foreach(DictionaryEntry datos in miTabla)

    Console.WriteLine("Key – {0}, Value –{1}", datos.Key,
            datos.Value);
```

III ÍNDICES VÁLIDOS

Para el **ArrayList** un índice se considera válido si no es menor que cero y es menor que la cantidad de elementos que contiene el **ArrayList**. Si se coloca un índice fuera de este rango se producirá un error. Si estos errores no se manejan adecuadamente, podemos tener pérdida de información o la finalización inesperada de la aplicación. Lo mejor es evitarlos.

Si lo deseamos podemos extraer solamente los valores y colocar una copia de ellos en una colección. Esto nos permitiría trabajar con los valores de una forma más parecida a lo que hemos aprendido anteriormente.

```
ICollection valores = miTabla.Values;
…
…
foreach(double valor in valores)
     Console.WriteLine("El valor es {0}",valor);
```

ICollection es una interfase usada para implementar las colecciones, de tal forma que valores puede actuar como cualquier colección válida que tengamos, en este caso la colección que representa los valores extraídos del **Hashtable**.

Para obtener un elemento del Hashtable

Si deseamos leer un elemento en particular del **Hashtable**, es posible que lo hagamos por medio de su propiedad **Item**. Para utilizarla, simplemente tenemos que colocar como índice el **key** que corresponde al valor que queremos leer en el momento. Debemos tener en cuenta que es posible que necesitemos hacer un type cast para dejar el valor correspondiente en el tipo necesario.

```
float valor;
…


…
valor = (float)miTabla.Item["Pan];
```

Para borrar los contenidos del Hashhtable

Todos los elementos guardados adentro del **Hashtable** pueden ser borrados al utilizar el método **Clear()**. Este método no necesita de ningún parámetro.

```
miTabla.Clear();
```

Para encontrar si ya existe un key

Como no podemos repetir el mismo **key** para dos elementos, es necesario poder saber si ya existe determinado **key** en el **Hashtable**. Poder hacerlo es fácil, pues C# nos provee del método **Contains()**. Este método necesita de un único parámetro que es el **key** a buscar y regresa un valor de tipo **bool**. Si el valor regresado es **true** significa que el **key** ya existe, y si es **false** que no se encuentra en el Hashtable.

```
bool exsite;
…
…
existe = miTabla.Contains("Pan");
```

Para encontrar si ya existe un value

Igualmente podemos buscar adentro del **Hashtable** por un **value** en particular. En este caso, usaremos el método **ContainsVale()**. Como parámetro colocaremos el valor a buscar y el método regresará un **bool**. Si el valor regresado es **true**, el **value** existe en el Hashtable y el valor de **false** nos indica que no se encuentra.

```
bool existe;
…
…
existe = miTabla.ContainsValue(17.50);
```

Para conocer la cantidad de parejas en el Hashtable

Si deseamos saber cuantas parejas **key-value** existen adentro de nuestro **Hashtable**, podemos usar la propiedad de **Count**. No hay que olvidar que el valor regresado es un entero que dice el número de parejas.

```
int cantidad;
…
…
cantidad = miTabla.Count;
```

Para eliminar un elemento del Hashtable

En el Hashtable no solamente podemos adicionar elementos, también podemos eliminarlos.

Al eliminarlos removemos la pareja **key-value** del **Hashtable**. Para poder llevar a cabo la eliminación, necesitamos conocer el **key** del valor a eliminar y utilizar el método **Remove()**. Este método necesita de un solo parámetro que es el **key** del elemento a borrar.

```
miTabla.Remove("Pan");
```

RESUMEN

Las colecciones nos permiten guardar elementos, a diferencia de los arreglos, éstas son dinámicas y pueden modificar su tamaño según sea necesario. Las colecciones tienen diferentes comportamientos, según la estructura de datos en la que estén basadas. Las que hemos aprendido en este capitulo son: ArrayList, Stack, Queue y Hashtable. Cada colección tiene su propia clase y necesitamos instanciarla para poder utilizarla. También nos proveen de diversos métodos para poder llevar a cabo las operaciones necesarias con los elementos a guardar.