Colecciones

Para muchas aplicaciones, puede que desee crear y administrar grupos de objetos relacionados.Existen dos formas de agrupar objetos: mediante la creación de matrices de objetos y con la creación de colecciones de objetos.

Las matrices son muy útiles para crear y trabajar con un número fijo de objetos fuertemente tipados.Para obtener información sobre las matrices, vea Matrices.

Las colecciones proporcionan una manera más flexible de trabajar con grupos de objetos. A diferencia de las matrices, el grupo de objetos con el que trabaja puede aumentar y reducirse de manera dinámica a medida que cambian las necesidades de la aplicación. Para algunas colecciones, puede asignar una clave a cualquier objeto que incluya en la colección para, de este modo, recuperar rápidamente el objeto con la clave.

Una colección es una clase, por lo que debe declarar una instancia de la clase para poder agregar elementos a dicha colección.

Si la colección contiene elementos de un solo tipo de datos, puede usar una de las clases del espacio de nombres System.Collections.Generic. Una colección genérica cumple la seguridad de tipos para que ningún otro tipo de datos se pueda agregar a ella. Cuando recupera un elemento de una colección genérica, no tiene que determinar su tipo de datos ni convertirlo.

Para los ejemplos de este tema, incluya las directivas **using** para los espacios de nombres `System.Collections.Generic` y `System.Linq`.

Uso de una colección Simple

Los ejemplos de esta sección usan la clase genérica List<T>, que le permite trabajar con una lista de objetos fuertemente tipados.

En el ejemplo siguiente se crea una lista de cadenas y luego se recorren en iteración mediante una instrucción foreach.

```
// Create a list of strings.
var salmons = new List<string>();
salmons.Add("chinook");
salmons.Add("coho");
salmons.Add("pink");
salmons.Add("sockeye");
```

```
// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Si el contenido de una colección se conoce de antemano, puede usar un inicializador de colecciónpara inicializar la colección. Para obtener más información, vea Inicializadores de objeto y colección.

El ejemplo siguiente es el mismo que el ejemplo anterior, excepto que se usa un inicializador de colección para agregar elementos a la colección.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

Puede usar una instrucción for en lugar de una instrucción foreach para recorrer en iteración una colección. Esto se consigue con el acceso a los elementos de la colección mediante la posición de índice. El índice de los elementos comienza en 0 y termina en el número de elementos menos 1.

El ejemplo siguiente recorre en iteración los elementos de una colección mediante for en lugar de foreach.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook coho pink sockeye
```

El ejemplo siguiente quita un elemento de la colección especificando el objeto que se quitará.

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye
```

El ejemplo siguiente quita elementos de una lista genérica. En lugar de una instrucción foreach, se usa una instrucción for que procesa una iteración en orden descendente. Esto es porque el método RemoveAt hace que los elementos después de un elemento quitado tengan un valor de índice inferior.

```
var numbers = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

Para el tipo de elementos de List<T>, también puede definir su propia clase. En el ejemplo siguiente, la clase Galaxy que usa List<T> se define en el código.

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
        {
            new Galaxy() { Name="Tadpole", MegaLightYears=400},
```

```
            new Galaxy() { Name="Pinwheel", MegaLightYears=25},
            new Galaxy() { Name="Milky Way", MegaLightYears=0},
            new Galaxy() { Name="Andromeda", MegaLightYears=3}
        };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Ordenar una colección

En el ejemplo siguiente se muestra un procedimiento para ordenar una colección. El ejemplo ordena las instancias de la clase `Car` que se almacenan en un List<T>. La clase `Car` implementa la interfaz IComparable<T>, que requiere implementar el método CompareTo.

Cada llamada al método CompareTo realiza una comparación única que se usa para la ordenación.El código escrito por el usuario en el método `CompareTo` devuelve un valor para cada comparación del objeto actual con otro objeto. El valor devuelto es menor que cero si el objeto actual es menor que el otro objeto, mayor que cero si el objeto actual es mayor que el otro objeto y cero si son iguales. Esto permite definir en el código los criterios de mayor que, menor que e igual.

En el método `ListCars`, la instrucción `cars.Sort()` ordena la lista. Esta llamada al método Sort de List<T> hace que se llame automáticamente al método `CompareTo` para los objetos `Car` de `List`.

```
private static void ListCars()
{
    var cars = new List<Car>
    {
```

```
        { new Car() { Name = "car1", Color = "blue", Speed = 20}},
        { new Car() { Name = "car2", Color = "red", Speed = 50}},
        { new Car() { Name = "car3", Color = "green", Speed = 10}},
        { new Car() { Name = "car4", Color = "blue", Speed = 50}},
        { new Car() { Name = "car5", Color = "blue", Speed = 30}},
        { new Car() { Name = "car6", Color = "red", Speed = 60}},
        { new Car() { Name = "car7", Color = "green", Speed = 50}}
    };

    // Sort the cars by color alphabetically, and then by speed
    // in descending order.
    cars.Sort();

    // View all of the cars.
    foreach (Car thisCar in cars)
    {
        Console.Write(thisCar.Color.PadRight(5) + " ");
        Console.Write(thisCar.Speed.ToString() + " ");
        Console.Write(thisCar.Name);
        Console.WriteLine();
    }

    // Output:
    //  blue  50 car4
    //  blue  30 car5
    //  blue  20 car1
    //  green 50 car7
    //  green 10 car3
    //  red   60 car6
    //  red   50 car2
}

public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
```

```
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

Definición de una colección personalizada

Puede definir una colección implementando la interfaz IEnumerable<T> o IEnumerable.

Aunque puede definir una colección personalizada, es mejor usar las colecciones incluidas en .NET Framework. Estas colecciones se describen en la sección Tipos de colecciones de este tema.

En el siguiente ejemplo se define una clase de colección personalizada denominada `AllColors`. Esta clase implementa la interfaz IEnumerable que requiere implementar el método GetEnumerator.

El método `GetEnumerator` devuelve una instancia de la clase `ColorEnumerator`. `ColorEnumerator`implementa la interfaz IEnumerator, que requiere que la propiedad Current, el método MoveNext y el método Reset estén implementados.

```
private static void ListColors()
{
    var colors = new AllColors();

    foreach (Color theColor in colors)
    {
        Console.Write(theColor.Name + " ");
    }
    Console.WriteLine();
    // Output: red blue green
}

// Collection class.
public class AllColors : System.Collections.IEnumerable
```

```
{
    Color[] _colors =
    {
        new Color() { Name = "red" },
        new Color() { Name = "blue" },
        new Color() { Name = "green" }
    };

    public System.Collections.IEnumerator GetEnumerator()
    {
        return new ColorEnumerator(_colors);

        // Instead of creating a custom enumerator, you could
        // use the GetEnumerator of the array.
        //return _colors.GetEnumerator();
    }

    // Custom enumerator.
    private class ColorEnumerator : System.Collections.IEnumerator
    {
        private Color[] _colors;
        private int _position = -1;

        public ColorEnumerator(Color[] colors)
        {
            _colors = colors;
        }

        object System.Collections.IEnumerator.Current
        {
            get
            {
                return _colors[_position];
            }
        }

        bool System.Collections.IEnumerator.MoveNext()
        {
            _position++;
            return (_position < _colors.Length);
        }

        void System.Collections.IEnumerator.Reset()
        {
            _position = -1;
        }
    }
}
```

```
// Element class.
public class Color
{
    public string Name { get; set; }
}
```

Iterators

Los *iteradores* se usan para efectuar una iteración personalizada en una colección. Un iterador puede ser un método o un descriptor de acceso `get`. Un iterador usa una instrucción yield returnpara devolver cada elemento de la colección a la vez.

Llame a un iterador mediante una instrucción foreach. Cada iteración del bucle `foreach` llama al iterador. Cuando se alcanza una instrucción `yield return` en el iterador, se devuelve una expresión y se conserva la ubicación actual en el código. La ejecución se reinicia desde esa ubicación la próxima vez que se llama al iterador.

El siguiente ejemplo usa el método del iterador. El método del iterador tiene una instrucción `yield return` que se encuentra dentro de un bucle for. En el método `ListEvenNumbers`, cada iteración del cuerpo de la instrucción `foreach` crea una llamada al método iterador, que continúa con la siguiente instrucción `yield return`.

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
```

```
}
```