Search MSDN with Bing

- MSDN Library
- Development Tools and Languages
- Visual Studio 2008
- Visual Studio
- Visual C#
- C# Programming Guide
  - Indexers (C# Programming Guide)
    - Using Indexers (C# Programming Guide)
    - Indexers in Interfaces (C# Programming G
    - Comparison Between Properties and Inde

Community Content

Add code samples and tips to enhance this topic.

More...

Using Indexers (C# Programming Guide)

Updated: March 2009

Indexers are a syntactic convenience that enable you to create a class, struct, or interface that client applications can access just as an array. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class named TempRecord that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period. The class contains an array named "temps" of type float to represent the temperatures, and a DateTime that represents the date the temperatures were recorded. By implementing an indexer in this class, clients can access the temperatures in a TempRecord instance as `float temp = tr[4]` instead of as `float temp = tr.temps[4]`. The indexer notation not only simplifies the syntax for client applications; it also makes the class and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the this keyword, as in this example:

```
public int this[int index]    // Indexer declaration
{
    // get and set accessors
}
```

Remarks

The type of an indexer and the type of its parameters must be at least as accessible as the indexer itself. For more information about accessibility levels, see Access Modifiers.

For more information about how to use indexers with an interface, see Interface Indexers.

The signature of an indexer consists of the number and types of its formal parameters. It does not include the indexer type or the names of the formal parameters. If you declare more than one indexer in the same class, they must have different signatures.

An indexer value is not classified as a variable; therefore, you cannot pass an indexer value as a ref or out parameter.

To provide the indexer with a name that other languages can use, use a `name` attribute in the declaration. For example:

```
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this [int index]   // Indexer declaration
{
}
```

This indexer will have the name `TheItem`. Not providing the name attribute would make `Item` the default name.

Example 1

Description

The following example shows how to declare a private array field, `temps`, and an indexer. The indexer enables direct access to the instance `tempRecord[i]`. The alternative to using the indexer is to declare the array as a public member and access its members, `tempRecord.temps[i]`, directly.

Notice that when an indexer's access is evaluated, for example, in a **Console.Write** statement, the get accessor is invoked. Therefore, if no **get** accessor exists, a compile-time error occurs.

Code

C#

```
class TempRecord
{
    // Array of temperature values
    private float[] temps = new float[10] { 56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
                                            61.3F, 65.9F, 62.1F, 59.2F, 57.5F };

    // To enable client code to validate input
    // when accessing your indexer.
    public int Length
    {
        get { return temps.Length; }
    }
    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
```

```
    {
        get
        {
            return temps[index];
        }

        set
        {
            temps[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        TempRecord tempRecord = new TempRecord();
        // Use the indexer's set accessor
        tempRecord[3] = 58.3F;
        tempRecord[5] = 60.1F;

        // Use the indexer's get accessor
        for (int i = 0; i < 10; i++)
        {
           System.Console.WriteLine("Element #{0} = {1}", i, tempRecord[i]);

        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();

    }
}
/* Output:
        Element #0 = 56.2
        Element #1 = 56.7
        Element #2 = 56.5
        Element #3 = 58.3
        Element #4 = 58.8
        Element #5 = 60.1
        Element #6 = 65.9
        Element #7 = 62.1
        Element #8 = 59.2
        Element #9 = 57.5
    */
```

Indexing Using Other Values

C# does not limit the index type to integer. For example, it may be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can co-exist.

Example 2

Description

In this example, a class is declared that stores the days of the week. A **get** accessor is declared that takes a string, the name of a day, and returns the corresponding integer. For example, Sunday will return 0, Monday will return 1, and so on.

Code

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = { "Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat" };

    // This method finds the day or returns -1
    private int GetDay(string testDay)
    {

        for(int j = 0; j < days.Length; j++)
        {
            if (days[j] == testDay)
            {
                return j;
            }
        }

        throw new System.ArgumentOutOfRangeException(testDay, "testDay must be in the
form \"Sun\", \"Mon\", etc");
```

```
        }

        // The get accessor returns an integer for a given string
        public int this[string day]
        {
            get
            {
                return (GetDay(day));
            }
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            DayCollection week = new DayCollection();
            System.Console.WriteLine(week["Fri"]);

            // Raises ArgumentOutOfRangeException
            System.Console.WriteLine(week["Made-up Day"]);

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
    // Output: 5
```

Robust Programming

There are two main ways in which the security and reliability of indexers can be improved:

- Be sure to incorporate some type of error-handling strategy to handle the chance of client code passing in an invalid index value. In the first example earlier in this topic, the TempRecord class provides a Length property that enables the client code to verify the input before passing it to the indexer. You can also put the error handling code inside the indexer itself. Be sure to document for users any exceptions that you throw inside an indexer accessor. For more information, see Design Guidelines for Exceptions.
- Set the accessibility of the **get** and set accessors to be as restrictive as is reasonable. This is important for the **set** accessor in particular. For more information, see Asymmetric Accessor Accessibility (C# Programming Guide).

See Also

Tasks

Indexers Sample

Concepts

C# Programming Guide

Reference

Indexers (C# Programming Guide)
Properties (C# Programming Guide)

Change History

Date	History	Reason
March 2009	Corrected an error in second example.	Customer feedback.

Community Content

Add

FAQ