Contents

Indexers (C# Programming Guide)

10/31/2018 • 2 minutes to read • Edit Online

Indexers allow instances of a class or struct to be indexed just like arrays. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with simple get and set accessor methods to assign and retrieve values. The `Program` class creates an instance of this class for storing strings.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get { return arr[i]; }
      set { arr[i] = value; }
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

NOTE

For more examples, see Related Sections.

Expression Body Definitions

It is common for an indexer's get or set accessor to consist of a single statement that either returns or sets a value. Expression-bodied members provide a simplified syntax to support this scenario. Starting with C# 6, a read-only indexer can be implemented as an expression-bodied member, as the following example shows.

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection.Add("Hello, World");
      System.Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Note that `=>` introduces the expression body, and that the `get` keyword is not used.

Starting with C# 7.0, both the get and set accessor can be an implemented as expression-bodied members. In this case, both `get` and `set` keywords must be used. For example:

```
using System;

class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}

class Program
{
   static void Main()
   {
      var stringCollection = new SampleCollection<string>();
      stringCollection[0] = "Hello, World.";
      Console.WriteLine(stringCollection[0]);
   }
}
// The example displays the following output:
//       Hello, World.
```

Indexers Overview

- Indexers enable objects to be indexed in a similar manner to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The this keyword is used to define the indexer.
- The value keyword is used to define the value being assigned by the `set` indexer.
- Indexers do not have to be indexed by an integer value; it is up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have more than one formal parameter, for example, when accessing a two-dimensional array.

Related Sections

- Using Indexers
- Indexers in Interfaces
- Comparison Between Properties and Indexers
- Restricting Accessor Accessibility

C# Language Specification

For more information, see Indexers in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See Also

- C# Programming Guide
- Properties

Using indexers (C# Programming Guide)

10/3/2018 • 4 minutes to read • Edit Online

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 `TempRecord` that represents the temperature in Farenheit as recorded at 10 different times during a 24 hour period. The class contains an array `temps` of type `float[]` to store the temperature values. 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 the following example shows:

```
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 doesn't 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 System.Runtime.CompilerServices.IndexerNameAttribute, as the following example shows:

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

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

Example 1

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.

```
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# doesn't 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

The following example declares a class that stores the days of the week. A `get` accessor takes a string, the name of a day, and returns the corresponding integer. For example, "Sunday" returns 0, "Monday" returns 1, and so on.

```
// 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.

- 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 Restricting Accessor Accessibility.

See also

- C# Programming Guide
- Indexers
- Properties

Indexers in Interfaces (C# Programming Guide)

9/4/2018 • 2 minutes to read • Edit Online

Indexers can be declared on an interface. Accessors of interface indexers differ from the accessors of class indexers in the following ways:

- Interface accessors do not use modifiers.
- An interface accessor does not have a body.

Thus, the purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only.

The following is an example of an interface indexer accessor:

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

Example

The following example shows how to implement interface indexers.

```
// Indexer on an interface:
public interface ISomeInterface
{
    // Indexer declaration:
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : ISomeInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        get
        {
            // The arr object will throw IndexOutOfRange exception.
            return arr[index];
        }
        set
        {
            arr[index] = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        IndexerClass test = new IndexerClass();
        System.Random rand = new System.Random();
        // Call the indexer to initialize its elements.
        for (int i = 0; i < 10; i++)
        {
            test[i] = rand.Next();
        }
        for (int i = 0; i < 10; i++)
        {
            System.Console.WriteLine("Element #{0} = {1}", i, test[i]);
        }

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
 */
```

In the preceding example, you could use the explicit interface member implementation by using the fully qualified name of the interface member. For example:

```
string ISomeInterface.this[int index]
{
}
```

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same indexer signature. For example, if an `Employee` class is implementing two interfaces, `ICitizen` and `IEmployee`, and both interfaces have the same indexer signature, the explicit interface member implementation is necessary. That is, the following indexer declaration:

```
string IEmployee.this[int index]
{
}
```

implements the indexer on the `IEmployee` interface, while the following declaration:

```
string ICitizen.this[int index]
{
}
```

implements the indexer on the `ICitizen` interface.

See Also

- C# Programming Guide
- Indexers
- Properties
- Interfaces

Comparison Between Properties and Indexers (C# Programming Guide)

9/4/2018 • 2 minutes to read • Edit Online

Indexers are like properties. Except for the differences shown in the following table, all the rules that are defined for property accessors apply to indexer accessors also.

PROPERTY	INDEXER
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A `get` accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit `value` parameter.	A `set` accessor of an indexer has the same formal parameter list as the indexer, and also to the value parameter.
Supports shortened syntax with Auto-Implemented Properties.	Does not support shortened syntax.

See Also

- C# Programming Guide
- Indexers
- Properties