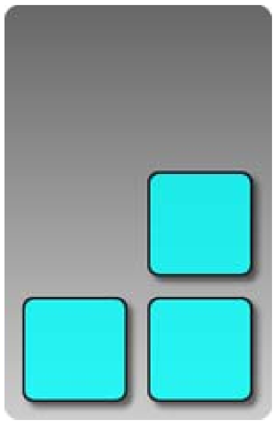
**Dot Net Perls** Sam Allen

# C# Dictionary Examples, Keys and Values

You want to use **Dictionary** in your C# program for constant lookup times and to associate keys with values. Look at some examples of using Dictionary with Keys and KeyValuePair, as well as with classes and methods. This document has tips and examples for using Dictionary with keys and values using the C# programming language.



> ***Dictionary*** *provides fast lookup of elements.*
> *Used when you have many different elements.*
> *Found in the System.Collections.Generic namespace.*
> *Specify the types of its keys and values.*

## Adding keys

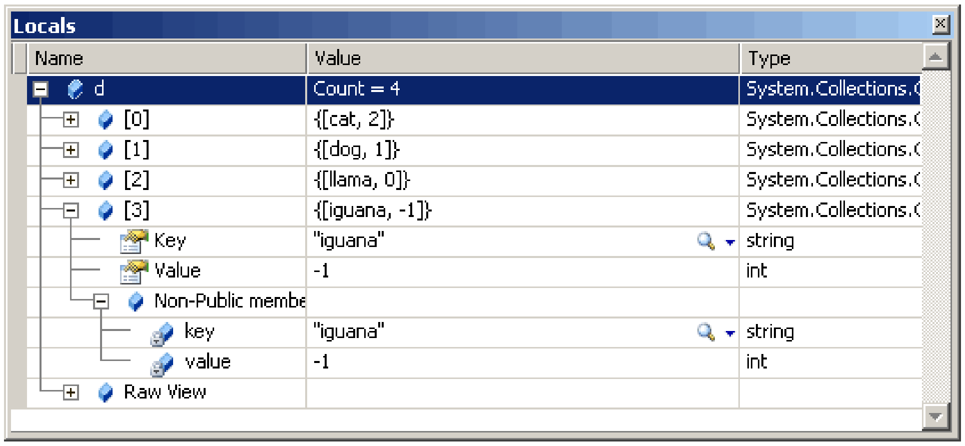
To get started, let's add four keys with values to a Dictionary instance. Afterwards, we look inside the Dictionary using Visual Studio's debugger. You will see that the Dictionary is composed of separate keys and values. Here's the Dictionary code and then its internal view.

**~~~ Program that uses Dictionary Add method (C#) ~~~**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Dictionary<string, int> d = new Dictionary<string, int>();
        d.Add("cat", 2);
        d.Add("dog", 1);
        d.Add("llama", 0);
        d.Add("iguana", -1);
    }
}
```

**Inside the Dictionary.** Here's what the above code looks like in memory. It is represented by a collection of key and value pairs. The screenshot is worth looking at. This article is based on .NET 3.5 SP1, but the contents apply to other versions as well.



Locals

| Name | Value | Type |
|---|---|---|
| d | Count = 4 | System.Collections.( |
| [0] | {[cat, 2]} | System.Collections.( |
| [1] | {[dog, 1]} | System.Collections.( |
| [2] | {[llama, 0]} | System.Collections.( |
| [3] | {[iguana, -1]} | System.Collections.( |
| Key | "iguana" | string |
| Value | -1 | int |
| Non-Public membe | | |
| key | "iguana" | string |
| value | -1 | int |
| Raw View | | |

## Looking up values

Here we see how you can check to see if a given string is present in a Dictionary with string keys. We look at more types of Dictionaries further on, but here is the ContainsKey method.

**=== Program that uses ContainsKey (C#) ===**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Dictionary<string, int> d = new Dictionary<string, int>();
        d.Add("apple", 1);
        d.Add("windows", 5);

        // See if Dictionary contains this string
        if (d.ContainsKey("apple")) // True
        {
            int v = d["apple"];
            Console.WriteLine(v);
        }

        // See if Dictionary contains this string
        if (d.ContainsKey("acorn"))
        {
            Console.WriteLine(false); // Nope
        }
    }
}
```

**=== Output of the program ===**

```
1
```

**Note on efficiency.** There is a more efficient method called TryGetValue on the Dictionary class, and you should definitely use it when possible. As its name implies, it tests for the key and then returns the value if it finds the key.
**See TryGetValue Method.**

## KeyNotFoundException

If you are running into the KeyNotFoundException, you are accessing a key in your Dictionary that doesn't exist. Dictionary is not the same as Hashtable and you must test keys for existence first, with ContainsKey or TryGetValue.
**See KeyNotFoundException Fix.**

## Understanding KeyValuePair

Hint: this is not in every beginner's C# book. When Dictionary, or any object that implements IDictionary, is used in a foreach loop, it returns an enumeration. In the case of Dictionary, this enumeration is in the form of KeyValuePairs.
**See KeyValuePair Collection Hints.**

## Using foreach on Dictionary

Here we use foreach syntax and KeyValuePair generics in the foreach loop. With collections like Dictionary, we must always know the value types. With each KeyValuePair, there is a Key member and Value member.

**=== Program that uses foreach on Dictionary (C#) ===**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Example Dictionary again
        Dictionary<string, int> d = new Dictionary<string, int>()
        {
            {"cat", 2},
            {"dog", 1},
            {"llama", 0},
            {"iguana", -1}
        };
        // Loop over pairs with foreach
        foreach (KeyValuePair<string, int> pair in d)
        {
            Console.WriteLine("{0}, {1}",
                pair.Key,
                pair.Value);
        }
        // Use var keyword to enumerate dictionary
        foreach (var pair in d)
        {
            Console.WriteLine("{0}, {1}",
                pair.Key,
                pair.Value);
        }
    }
}
```

**=== Output of the program ===**

```
cat, 2
dog, 1
llama, 0
iguana, -1

cat, 2
dog, 1
llama, 0
iguana, -1
```

**Overview of the code.** The code example declares and populates an example Dictionary. This Dictionary happens to indicate what animals we have and how many of them.

**Using the foreach loop.** It has a ShowDictionaryPair method. This method demonstrates the foreach loop and the KeyValuePair declaration. Pay careful attention to the syntax in the foreach loop. Each KeyValuePair has two members, pair.Key and pair.Value, which contain string keys and int values.

**Using the var keyword.** The final loop in the code shows how you can make the syntax for looping really simple by using the var keyword. This is not always desirable on some projects, however.

## Getting all Dictionary keys

Here we use the Keys property and then look through each key and lookup the values. This method is slower but has the same results. Using the Keys collection and putting it in an array or List is very effective in other situations.

**=== Program that gets Keys from Dictionary (C#) ===**

```
using System;
using System.Collections.Generic;
```

```
class Program
{
    static void Main()
    {
        Dictionary<string, int> d = new Dictionary<string, int>()
        {
            {"cat", 2},
            {"dog", 1},
            {"llama", 0},
            {"iguana", -1}
        };
        // Store keys in a List
        List<string> list = new List<string>(d.Keys);
        // Loop through list
        foreach (string k in list)
        {
            Console.WriteLine("{0}, {1}",
                k,
                d[k]);
        }
    }
}
```

```
=== Output of the program ===

cat, 2
dog, 1
llama, 0
iguana, -1
```

## Benchmarking KeyValuePair usage

Using foreach on KeyValuePairs is several times faster than using Keys. This is probably because the Keys collection is not used. KeyValuePair allows us to simply look through each pair one at a time. This avoids lookups and using the garbage-collected heap for storage.
**See Garbage Collection Visualizations.**

```
=== Benchmark for KeyValuePair foreach loop ===

KeyValuePair: 125 ms
Note:         This loops through the pairs in the Dictionary.

Keys loop:    437 ms
Note:         This gets the Keys, then loops through them.
              It does another lookup for the value.
```

**Note on the benchmark.** The author made a small change to the Keys version for clarity and performance, so these figures are only general and apply to the previous version. Using KeyValuePair is most likely still faster.

## Sorting your Dictionary values

If you need to sort the values in your Dictionary, you may be perplexed at first and wonder how to order the keys properly. Fortunately, I have an article about how to do this, although it is not optimal.
**See Sort Dictionary Values.**

## Related collections

You will find other collections, such as SortedDictionary, in the base class libraries, BCL, available for you to use. However, my experience is that it is hard to get as good performance as with Dictionary.
**See SortedDictionary.**

## Using different types in Dictionary

Dictionary in C# is a generic class, which means it requires you to specify a type for it to use. So, you can use an int key, just as easily as a string key. Here is an example of a Dictionary with int keys.

**=== Program that uses int keys (C#) ===**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // Use a dictionary with an int key.
        Dictionary<int, string> dict = new Dictionary<int, string>();
        dict.Add(100, "Bill");
        dict.Add(200, "Steve");
        // You can lookup the int in the dictionary.
        if (dict.ContainsKey(200))
        {
            Console.WriteLine(true);
        }
    }
}
```

**=== Output of the program ===**

```
True
```

**Advanced features of Dictionary.** For more advanced developers, you can use the GetHashCode() method and override it to create Dictionaries or hashes with the class. This can improve performance in those cases.

## Enhancing lookup speed

To enhance lookup speed on your Dictionary, you can change the size of the keys you use. My research has shown that when you use shorter string keys, the lookup time is improved. This could create more collisions, so testing may be necessary.

## Using Dictionary with LINQ

By using the ToDictionary method, which is an extension method on IEnumerable that will place the keys and values into a new Dictionary using lambda expressions.

**=== Program that uses LINQ (C#) ===**

```
using System;
using System.Collections.Generic;
using System.Linq;

class Program
{
    static void Main()
    {
        string[] arr = new string[]
        {
            "One",
            "Two"
        };
        var dict = arr.ToDictionary(item => item, item => true);
        foreach (var pair in dict)
        {
            Console.WriteLine("{0}, {1}",
                pair.Key,
                pair.Value);
        }
    }
}
```

**=== Output of the program ===**

```
One, True
Two, True
```

**Explanation of the example.** The above example uses ToDictionary, which resides in the System.Linq namespace, on the string[] array to create a lookup table where both strings can be accessed in constant time, O(1).

## Using ContainsValue to find values

Dictionary also helpfully implements a method called ContainsValue. This method does not enjoy the constant-time lookup speed that ContainsKey has, however. It instead searches the entire collection, making it linear in complexity.

**=== Program that uses ContainsValue (C#) ===**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Dictionary<string, int> d = new Dictionary<string, int>();
        d.Add("cat", 1);
        d.Add("dog", 2);
        if (d.ContainsValue(1))
        {
            Console.WriteLine(true); // true
        }
    }
}
```

**=== Output of the program ===**

```
True
```

**ContainsValue method.** The above example will internally loop through all elements in the Dictionary until it finds the match, or there are no more to check. MSDN: "This method is an O(n) operation, where n is Count."
**See ContainsValue Dictionary Example.**

## Clearing and counting

You can erase all the key/value pairs within your Dictionary by using the Clear() method, which accepts no parameters. Alternatively, you can assign the variable to null. The difference between Clear and null is not important for memory, as in either case the entries are garbage-collected. Internally, I see that Clear calls Array.Clear, which is not managed code.

**Counting your Dictionary.** The Count method on the Dictionary collection gives you an effective way of computing the total number of keys in the instance. This is much simpler than accessing the Keys property or looping over the Dictionary to count it. This Count property is covered in more detail on this site.
**See Count Dictionary, Using Count Property.**

## Removing an entry

Here you want to eliminate an entry, not just by setting its value to null or string.Empty, but by also removing the key itself. Fortunately, you can use the Remove method.

**~~~ Program that uses Remove (C#) ~~~**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Dictionary<string, int> d = new Dictionary<string, int>();
        d.Add("cat", 1);
        d.Add("dog", 2);

        d.Remove("cat");     // Removes cat
        d.Remove("nothing"); // Doesn't remove anything
    }
}
```

**Execution of the code.** Running the code in Visual Studio, no exceptions are thrown, which means that when you remove a key that doesn't exist, nothing happens. However, Remove throws System.ArgumentNullException when it receives a null parameter.

## Copying entire Dictionary

The Dictionary class has a useful constructor that allows you to easily copy all the values and keys in your Dictionary into a new Dictionary instance. You can write the logic yourself, but using this constructor improves code reuse and simplicity. This site has more information on the Dictionary copy constructor.
**See Copy Dictionary.**

## Parameters and return values

It is also possible to use the Dictionary constructed type in the C# language as a parameter to methods or as a return value from methods or properties. Because the Dictionary type is defined as a class, it is always passed as a reference type, meaning only 32-64 bits will be copied on the method invocation. The same principles apply when copying a Dictionary return value from a method. You can find more information on Dictionary parameters on this site.
**See Dictionary Parameter and Return Value.**

## Should I use List or Dictionary?

I suggest you almost always use Dictionary when you need to do lookups. If you use List and you need to look up a key, your program may freeze if you happen to have a huge number of elements. In other words, if you use Dictionary, your program can recover from pathological, edge cases.

## Using multiple variables in single key

You can sometimes use multiple variables in a key by creating a special function that transforms those variables into a string, serializing them. So, you could use the string "1,2" to mean the ints 1 and 2.

## Initializing Dictionary at class level

Sometimes it is useful to have a Dictionary in your class that is allocated at the class level, not in a method or constructor. Additionally, if you have a static class then you should always initialize your Dictionary at the class level like this instead of the static constructor. Static constructors have performance penalties, which I have measured.

**=== Program that uses Dictionary with class (C#) ===**

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Example e = new Example();
        Console.WriteLine(e.GetValue());
    }
}

class Example
{
    Dictionary<int, int> _d = new Dictionary<int, int>()
    {
        {1, 1},
        {2, 3}, // <--
        {3, 5},
        {6, 10}
    };
    public int GetValue()
    {
        return _d[2]; // Example only
    }
}
```

**=== Output of the program ===**

```
3
```

## Summary

In this article, we saw how you can use Dictionary with KeyValuePair to look through all pairs in the Dictionary. Additionally, we saw material on ContainsKey, which lets you check key existence. Finally, the author considers Dictionary one of the most interesting subjects in the programming world.

**See Dictionary Articles.**