

# What is more efficient: Dictionary TryGetValue or ContainsKey+Item?

From MSDN's entry on [Dictionary.TryGetValue Method](#):

233

This method combines the functionality of the ContainsKey method and the Item property.

If the key is not found, then the value parameter gets the appropriate default value for the value type TValue; for example, 0 (zero) for integer types, false for Boolean types, and null for reference types.



34

Use the TryGetValue method if your code frequently attempts to access keys that are not in the dictionary. Using this method is more efficient than catching the KeyNotFoundException thrown by the Item property.

This method approaches an  $O(1)$  operation.

From the description, it's not clear if it is more efficient or just more convenient than calling ContainsKey and then doing the lookup. Does the implementation of TryGetValue just call ContainsKey and then Item or is actually more efficient than that by doing a single lookup?

In other words, what is more efficient (i.e. which one performs less lookups):

```
Dictionary<int,int> dict;
//...//
int ival;
if(dict.ContainsKey(ikey))
{
    ival = dict[ikey];
}
else
{
    ival = default(int);
}
```

or

```
Dictionary<int,int> dict;
//  //
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).


Note: I am not looking for a benchmark!

c#


performance

dictionary

edited Jun 19 '15 at 14:36

 **vaxquis**  
8,096 5 40 58

asked Feb 21 '12 at 18:01

 **Rado**  
5,245 6 26 42

10 Answers

▲

289

▼


✓

TryGetValue will be faster.


ContainsKey uses the same check as TryGetValue, which internally refers to the actual entry location. The Item property actually has nearly identical code functionality as TryGetValue, except that it will throw an exception instead of returning false.

Using ContainsKey followed by the Item basically duplicates the lookup functionality, which is the bulk of the computation in this case.

edited Mar 1 at 16:28

 **Lucas**  
15.5k 5 39 40

answered Feb 21 '12 at 18:06

 **Reed Copsey**  
479k 60 1002 1287

- 1

This is more subtle: `if(dict.ContainsKey(ikey)) dict[ikey]++; else dict.Add(ikey, 0);`. But i think that TryGetValue is still more efficient since the get *and* set of the indexer property is used, isn't it? – Tim Schmelter Oct 1 '15 at 10:44
- 3

you can actually look at the .net source for it now too: [referencesource.microsoft.com/#mscorlib/system/collections/...](#) you can see that all 3 of TryGetValue, ContainsKey, and this[] call the same FindEntry method and do the same amount of work, only differing in how they answer the question: trygetvalue returns bool and the value, contains key only returns true/false, and this[] returns the value or throws an exception. – John Gardner Jun 2 '16 at 17:12
- 1

@JohnGardner Yes, which is what I said - but if you do ContainsKey then get Item, you're doing that work 2x instead of 1x. – Reed Copsey Jun 2 '16 at 18:45
- 3

i agree completely :) i was just pointing out that the actual source is available now. none of the other answers/etc had a link to the actual source :D – John Gardner Jun 3 '16 at 17:32
- 1

Slightly off topic, if you're accessing via an IDictionary in a multithreaded environment I would always use TryGetValue as the state may change from

A quick benchmark shows that `TryGetValue` has a slight edge:

85

```
static void Main() {
    var d = new Dictionary<string, string> {{"a", "b"}};
    var start = DateTime.Now;
    for (int i = 0; i != 10000000; i++) {
        string x;
        if (!d.TryGetValue("a", out x)) throw new ApplicationException("Oops");
        if (d.TryGetValue("b", out x)) throw new ApplicationException("Oops");
    }
    Console.WriteLine(DateTime.Now-start);
    start = DateTime.Now;
    for (int i = 0; i != 10000000; i++) {
        string x;
        if (d.ContainsKey("a")) {
            x = d["a"];
        } else {
            x = default(string);
        }
        if (d.ContainsKey("b")) {
            x = d["b"];
        } else {
            x = default(string);
        }
    }
}
```

This produces

```
00:00:00.7600000
00:00:01.0610000
```

making the `ContainsKey + Item` access about 40% slower assuming an even blend of hits and misses.

Moreover, when I change the program to always miss (i.e. always looking up `"b"`) the two versions become equally fast:

```
00:00:00.2850000
00:00:00.2720000
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

00:00:00.4930000  
00:00:00.8110000

edited Feb 21 '12 at 18:16

answered Feb 21 '12 at 18:08



dasblinkenlight

627k 63 842 1250

- 
- 10 Of course, it depends on the actual usage pattern. If you almost never fail a lookup then `TryGetValue` should be far ahead. Also... a nitpick... `DateTime` is not the best way to capture performance measurements. – [Ed S.](#) Feb 21 '12 at 18:10
- 
- 3 @EdS. You are right, `TryGetValue` gets even further into the lead. I edited the answer to include an "all hits" and "all misses" scenarios. – [dasblinkenlight](#) Feb 21 '12 at 18:17
- 
- 2 @Luciano explain how you used `Any` - Like this: `Any(i=>i.Key==key)` . In which case, yes, that's a bad linear search of the dictionary. – [weston](#) Dec 4 '12 at 12:18
- 
- 11 `DateTime.Now` will only be accurate to a few ms. Use the `Stopwatch` class in `System.Diagnostics` instead (which uses `QueryPerformanceCounter` under the covers to provide much higher accuracy). It's easier to use, too. – [Alastair Maw](#) Jan 16 '13 at 11:05
- 
- 5 In addition to Alastair and Ed's comments - `DateTime.Now` can go backwards, if you get a time update, such as that which occurs when the user updates their computer time, a time zone is crossed, or the time zone changes (DST, for instance). Try working on a system that has the system clock synced to time provided by some radio service like GPS or mobile phone networks. `DateTime.Now` will go all over the place, and `DateTime.UtcNow` only fixes one of those causes. Just use `StopWatch`. – [antiduh](#) Feb 14 '14 at 22:38
- 

Since none of the answers thus far actually answer the question, here is an acceptable answer I found after some research:

46

If you decompile `TryGetValue` you see that it's doing this:

```
public bool TryGetValue(TKey key, out TValue value)
{
    int index = this.FindEntry(key);
    if (index >= 0)
    {
        value = this.entries[index].value;
        return true;
    }
    value = default(TValue);
    return false;
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

whereas the ContainsKey method is:

```
public bool ContainsKey(TKey key)
{
    return (this.FindEntry(key) >= 0);
}
```

so TryGetValue is just ContainsKey plus an array lookup if the item is present.

[Source](#)

It appears that TryGetValue will be almost twice as fast as ContainsKey+Item combination.

answered Feb 21 '12 at 18:59



Rado

5,245

6

26

42

Who cares :-)

18

You're probably asking because TryGetValue is a pain to use - so encapsulate it like this with an extension method.

```
public static class CollectionUtils
{
    // my original method
    // public static V GetValueOrDefault<K, V>(this Dictionary<K, V> dic, K key)
    // {
    //     V ret;
    //     bool found = dic.TryGetValue(key, out ret);
    //     if (found)
    //     {
    //         return ret;
    //     }
    //     return default(V);
    // }

    // EDIT: one of many possible improved versions
    public static TValue GetValueOrDefault<K, V>(this IDictionary<K, V> dictionary, K
    key)
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

TValue value;

// attempt to get the value of the key from the dictionary
dictionary.TryGetValue(key, out value);
return value;
}

```

Then just call :

```
dict.GetValueOrDefault("keyname")
```

or

```
(dict.GetValueOrDefault("keyname") ?? fallbackValue)
```

edited Dec 9 '17 at 10:51

answered May 12 '15 at 23:52



[Simon\\_Weaver](#)

77.7k 65 481 544

- 1 @Hüseyin I got very confused how I was stupid enough to post this without this but it turns out I have my method duplicated twice in my code base - once with and one without the this so that's why I never caught it! thanks for fixing! – [Simon\\_Weaver](#) Jul 15 '16 at 21:35 ✎
  - 2 TryGetValue assigns a default value to the out value parameter if the key doesnt exist, so this could be simplified. – [Raphael Smit](#) Aug 27 '16 at 17:24 ✎
  - 2 Simplified version: public static TValue GetValueOrDefault<TKey, TValue>(this Dictionary<TKey, TValue> dict, TKey key) { TValue ret; dict.TryGetValue(key, out ret); return ret; } – [Joshua](#) Oct 24 '16 at 20:00 ✎
  - 2 In C#7 this is really fun: if(!dic.TryGetValue(key, out value item)) item = dic[key] = new Item(); – [Shimmy](#) May 26 '17 at 7:21
- This could confuse even more people at your company, now they will have 3 options TryGetValue , ContainsKey , or GetValueOrDefault – [Jaider](#) Oct 27 '17 at 3:00 ✎



Why don't you test it?

10

But I'm pretty sure that TryGetValue is faster, because it only does one lookup. Of course this isn't guaranteed, i.e. different implementations might have different performance characteristics

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

The way I'd implement a dictionary is by creating an internal `Find` function that finds the slot for an item, and then build the rest on top of that.

answered Feb 21 '12 at 18:04



[CodesInChaos](#)

91.5k 14 176 233

---

I don't think the implementation details can possibly change the guarantee that doing action X once is faster than or equal to doing action X twice. Best case they're identical, worse case the 2X version takes twice as long. – [Dan Bechard](#) Jun 8 '17 at 18:48 ✎

---



All of the answers so far, although good, miss a vital point.

10



Methods into the classes of an API (e.g. the .NET framework) form part of an interface definition (not a C# or VB interface, but an interface in the computer science meaning).

As such, it is usually incorrect to ask whether calling such a method is faster, unless speed is a part of the formal interface definition (which it isn't in this case).

Traditionally this kind of shortcut (combining search and retrieve) is more efficient regardless of language, infrastructure, OS, platform, or machine architecture. It is also more readable, because it expresses your intent explicitly, rather than implying it (from the structure of your code).

So the answer (from a grizzled old hack) is definitely 'Yes' (TryGetValue is preferable to a combination of ContainsKey and Item [Get] to retrieve a value from a Dictionary).

If you think this sounds odd, think of it like this: Even if current implementations of TryGetValue, ContainsKey, and Item [Get] do not yield any speed difference, you can assume it is likely that a future implementation (e.g. .NET v5) will do (TryGetValue will be faster). Think about the lifetime of your software.

As an aside, it is interesting to note that typical modern interface definition technologies still rarely provide any means of formally defining timing constraints. Maybe .NET v5?


answered Dec 31 '15 at 14:00



[debater](#)

413 5 9

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

suboptimal implementation such that the semantically correct thing happens to be slower, unless you do the test. – [Dan Bechard](#) Jun 8 '17 at 18:52 

Making a quick test program, there is definately an improvement using TryGetValue with 1 million items in a dictionary.

5

## Results:

ContainsKey + Item for 1000000 hits: 45ms

TryGetValue for 1000000 hits: 26ms

Here is the test app:

```
static void Main(string[] args)
{
    const int size = 1000000;

    var dict = new Dictionary<int, string>();

    for (int i = 0; i < size; i++)
    {
        dict.Add(i, i.ToString());
    }

    var sw = new Stopwatch();
    string result;

    sw.Start();

    for (int i = 0; i < size; i++)
    {
        if (dict.ContainsKey(i))
            result = dict[i];
    }

    sw.Stop();
    Console.WriteLine("ContainsKey + Item for {0} hits: {1}ms", size,
sw.ElapsedMilliseconds);

    sw.Reset();
    sw.Start();
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).



```

    }

    sw.Stop();
    Console.WriteLine("TryGetValue for {0} hits: {1}ms", size, sw.ElapsedMilliseconds);
}

```

edited Feb 21 '12 at 18:11

answered Feb 21 '12 at 18:09



Brandon

59k 28 179 212



davisoa

4,700 20 33

On my machine, with loads of RAM, when run in RELEASE mode (not DEBUG), `ContainsKey` equals `TryGetValue` / `try-catch` if all entries in the `Dictionary<>` are found.

4

`ContainsKey` outperforms them all by far when there are just a few dictionary entries not found (in my example below, set `MAXVAL` to anything larger than `ENTRIES` to have some entries missed):

### Results:

```

Finished evaluation .... Time distribution:
Size: 000010: TryGetValue: 53,24%, ContainsKey: 1,74%, try-catch: 45,01% - Total:
2.006,00
Size: 000020: TryGetValue: 37,66%, ContainsKey: 0,53%, try-catch: 61,81% - Total:
2.443,00
Size: 000040: TryGetValue: 22,02%, ContainsKey: 0,73%, try-catch: 77,25% - Total:
7.147,00
Size: 000080: TryGetValue: 31,46%, ContainsKey: 0,42%, try-catch: 68,12% - Total:
17.793,00
Size: 000160: TryGetValue: 33,66%, ContainsKey: 0,37%, try-catch: 65,97% - Total:
36.840,00
Size: 000320: TryGetValue: 34,53%, ContainsKey: 0,39%, try-catch: 65,09% - Total:
71.059,00
Size: 000640: TryGetValue: 32,91%, ContainsKey: 0,32%, try-catch: 66,77% - Total:
141.789,00
Size: 001280: TryGetValue: 39,02%, ContainsKey: 0,35%, try-catch: 60,64% - Total:
244.657,00
Size: 002560: TryGetValue: 35,48%, ContainsKey: 0,19%, try-catch: 64,33% - Total:
420.121,00
Size: 005120: TryGetValue: 43,41%, ContainsKey: 0,24%, try-catch: 56,34% - Total:
625.969,00

```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

2.405.821,00

Size: 040960: TryGetValue: 37,28%, ContainsKey: 0,24%, try-catch: 62,48% - Total:

4.200.839,00

Size: 081920: TryGetValue: 29,68%, ContainsKey: 0,54%, try-catch: 69,77% - Total:

8.980.230,00

## Here's my code:

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

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            const int ENTRIES = 10000, MAXVAL = 15000, TRIALS = 100000, MULTIPLIER =
2;

            Dictionary<int, int> values = new Dictionary<int, int>();
            Random r = new Random();
            int[] lookups = new int[TRIALS];
            int val;
            List<Tuple<long, long, long>> durations = new List<Tuple<long, long,
long>>(8);

            for (int i = 0; i < ENTRIES; ++i) try
            {
                values.Add(r.Next(MAXVAL), r.Next());
            }
            catch { --i; }

            for (int i = 0; i < TRIALS; ++i) lookups[i] = r.Next(MAXVAL);

            Stopwatch sw = new Stopwatch();
            ConsoleColor bu = Console.ForegroundColor;

            for (int size = 10; size <= TRIALS; size *= MULTIPLIER)
            {
                long a, b, c;

                Console.ForegroundColor = ConsoleColor.Yellow;
                Console.WriteLine("Loop size: {0}", size);
```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```

----
        sw.Start();
        for (int i = 0; i < size; ++i) values.TryGetValue(lookups[i], out
val);
        sw.Stop();
        Console.WriteLine("TryGetValue: {0}", a = sw.ElapsedTicks);

        // -----

        sw.Restart();
        for (int i = 0; i < size; ++i) val = values.ContainsKey(lookups[i]) ?
values[lookups[i]] : default(int);
        sw.Stop();
        Console.WriteLine("ContainsKey: {0}", b = sw.ElapsedTicks);

        // -----

        sw.Restart();
        for (int i = 0; i < size; ++i)
            try { val = values[lookups[i]]; }
            catch { }
        sw.Stop();
        Console.WriteLine("try-catch: {0}", c = sw.ElapsedTicks);

        // -----

        Console.WriteLine();

        durations.Add(new Tuple<long, long, long>(a, b, c));
    }

    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine("Finished evaluation .... Time distribution:");
    Console.ForegroundColor = bu;

    val = 10;
    foreach (Tuple<long, long, long> d in durations)
    {
        long sum = d.Item1 + d.Item2 + d.Item3;

        Console.WriteLine("Size: {0:D6}:", val);
        Console.WriteLine("TryGetValue: {0:P2}, ContainsKey: {1:P2}, try-
catch: {2:P2} - Total: {3:N}", (decimal)d.Item1 / sum, (decimal)d.Item2 / sum,
(decimal)d.Item3 / sum, sum);
        val *= MULTIPLIER;
    }

```

By using our site, you acknowledge that you have read and understand our Cookie Policy, Privacy Policy, and our Terms of Service.

```
    }
}
```

answered Jan 21 '15 at 23:41



AxD

636

7

20

How about some explanation of your code? – [ViRuSTriNiTy](#) Feb 5 '16 at 8:02

I feel like something fishy is going on here. I wonder if the optimizer might be removing or simplifying your ContainsKey() checks due to the fact that you never use the retrieved value. – [Dan Bechard](#) Jun 8 '17 at 18:55

It just can't. ContainsKey() is in a compiled DLL. The optimizer doesn't know anything about what ContainsKey() actually does. It might cause side effects, so it has to be called and cannot be abridged. – [AxD](#) Jun 8 '17 at 22:24

Something is bogus here. The fact is that examining the .NET code shows that ContainsKey, TryGetValue, and this[] all call the same internal code, so TryGetValue is faster than ContainsKey + this[] when the entry exists. – [Jim Balter](#) Dec 9 '17 at 10:29

2

If you're trying to get out the value from the dictionary, the TryGetValue(key, out value) is the best option, but if you're checking for the presence of the key, for a new insertion, without overwriting old keys, and only with that scope, ContainsKey(key) is the best option, benchmark can confirm this:

```
using System;
using System.Threading;
using System.Diagnostics;
using System.Collections.Generic;
using System.Collections;

namespace benchmark
{
    class Program
    {
        public static Random m_Rand = new Random();
        public static Dictionary<int, int> testdict = new Dictionary<int, int>();
        public static Hashtable testhash = new Hashtable();

        public static void Main(string[] args)
        {
            Console.WriteLine("Adding elements into hashtable...");
        }
    }
}
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

watch.Stop();
Console.WriteLine("Done in {0:F4} -- pause....", watch.Elapsed.TotalSeconds);
Thread.Sleep(4000);
Console.WriteLine("Adding elements into dictionary...");
watch = Stopwatch.StartNew();
for(int i=0; i<1000000; i++)
    testdict[i]=m_Rand.Next();
watch.Stop();
Console.WriteLine("Done in {0:F4} -- pause....", watch.Elapsed.TotalSeconds);
Thread.Sleep(4000);

Console.WriteLine("Finding the first free number for insertion");
Console.WriteLine("First method: ContainsKey");
watch = Stopwatch.StartNew();
int intero=0;
while (testdict.ContainsKey(intero))
{
    intero++;
}
testdict.Add(intero, m_Rand.Next());
watch.Stop();
Console.WriteLine("Done in {0:F4} -- added value {1} in dictionary --
pause....", watch.Elapsed.TotalSeconds, intero);
Thread.Sleep(4000);
Console.WriteLine("Second method: TryGetValue");
watch = Stopwatch.StartNew();
intero=0;
int result=0;
while(testdict.TryGetValue(intero, out result))
{
    intero++;
}
testdict.Add(intero, m_Rand.Next());
watch.Stop();
Console.WriteLine("Done in {0:F4} -- added value {1} in dictionary --
pause....", watch.Elapsed.TotalSeconds, intero);
Thread.Sleep(4000);
Console.WriteLine("Test hashtable");
watch = Stopwatch.StartNew();
intero=0;
while(testhash.Contains(intero))
{
    intero++;
}
testhash.Add(intero, m_Rand.Next());
watch.Stop();

```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

```

        Console.ReadKey(true);
    }
}

```

This is a true Example, I have a service that for each "Item" created, it associates a progressive number, this number, each time you create a new item, must be found free, if you delete an Item, the free number becomes free, of course this is not optimized, since I have a static var that caches the current number, but in case you end all the numbers, you can re-begin from 0 to UInt32.MaxValue

Test executed:

Adding elements into hashtable...

Done in 0,5908 -- pause....

Adding elements into dictionary...

Done in 0,2679 -- pause....

Finding the first free number for insertion

First method: ContainsKey

Done in 0,0561 -- added value 1000000 in dictionary -- pause....

Second method: TryGetValue

Done in 0,0643 -- added value 1000001 in dictionary -- pause....

Test hashtable

Done in 0,3015 -- added value 1000000 into hashtable -- pause....

Press any key to continue . .

If some of you may be asking if the ContainsKeys could have an advantage, I've even tried inverting the TryGetValue with Contains key, the result is the same.

So, for me, with a final consideration, it all depends on the way the program behaves.

edited Jul 21 '13 at 6:12

answered Jul 21 '13 at 5:43



Fwiffo

21 2



2

Apart from designing a microbenchmark that will give accurate results in a practical setting, you can inspect the reference source of .NET Framework.

- [System.Collections.Generic.Dictionary<TKey, TValue>.TryGetValue\(TKey, out TValue\)](#).

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).

- [System.Collections.Generic.Dictionary<TKey, TValue>.Item\(TKey\)](#).

All of them call the [FindEntry\(TKey\)](#) method that does most of the work and does not memoize its result, so **calling TryGetValue is almost twice as fast as** `ContainsKey + Item`.

The inconvenient interface of `TryGetValue` can be **adapted using an extension method**:

```
using System.Collections.Generic;

namespace Project.Common.Extensions
{
    public static class DictionaryExtensions
    {
        public static TValue GetValueOrDefault<TKey, TValue>(
            this IDictionary<TKey, TValue> dictionary,
            TKey key,
            TValue defaultValue = default(TValue))
        {
            if (dictionary.TryGetValue(key, out TValue value))
            {
                return value;
            }
            return defaultValue;
        }
    }
}
```

Since C# 7.1, you can replace `default(TValue)` with plain `default`. [The type is inferred](#).

Usage:

```
var dict = new Dictionary<string, string>();
string val = dict.GetValueOrDefault("theKey", "value used if theKey is not found in dict");
```

It returns `null` for reference types whose lookup fails, unless an explicit default value is specified.

```
var dictObj = new Dictionary<string, object>();
object valObj = dictObj.GetValueOrDefault("nonexistent");
```

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).


```
int valInt = dictInt.GetValueOrDefault("nonexistent");  
Debug.Assert(valInt == 0);
```

answered May 25 '18 at 7:56

[Palec](#)

8,074 5 43 92

---

Note that users of the extension method can't tell the difference between a non-existent key and a key that exists but its value is default(T). – [Lucas](#) Feb 27 at 18:46 

---

By using our site, you acknowledge that you have read and understand our [Cookie Policy](#), [Privacy Policy](#), and our [Terms of Service](#).