ConcurrentDictionary vs Hashtable.Synchronized

Comparison of collections in multithread environment

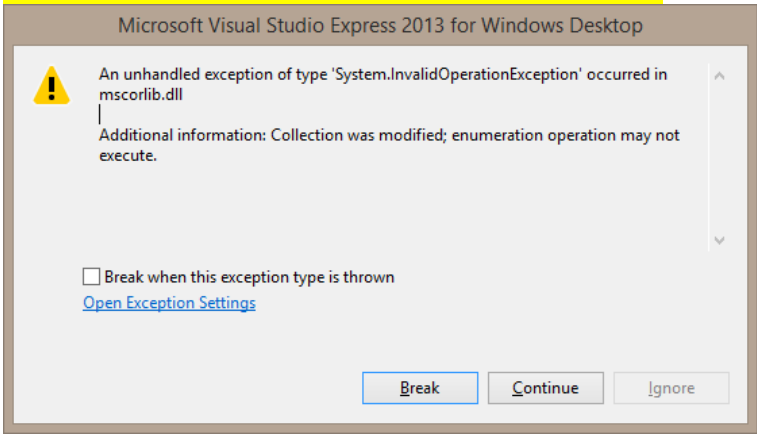
Basically all Dictionary collections in .NET internally implement Hashtable wrapped, so in terms of mechanism they are pretty much the same. The difference is that Dictionary exposes strong types while Hashtable uses object-object keyvalue pair which is much harder to work with because you need to know what is the actual type of the object stored in a key or value. Often casting will probably be required.

In multithread code environment locking is considered by default when accessing any shared resource.
For most of the time this is some kind of object or collection.
Collections are tricky because you will run into and exception if one thread reads collection while the other one is modifying.

```
1.  using System;
2.  using System.Collections;
3.  using System.Collections.Concurrent;
4.  using System.Collections.Generic;
5.  using System.Threading;
6.
7.  namespace ThreadCollection
8.  {
9.      class ProgramHashtable
10.     {
11.         static Hashtable sharedList = Hashtable.Synchronized(new Hashtable());
12.         static void Main(string[] args)
13.         {
14.             //Initial load of 10 elements
15.             for (int i = 0; i < 10; i++)
16.             {
17.                 sharedList.Add(Guid.NewGuid(), new object());
18.             }
19.
20.             //Reading thread
21.             new Thread(() =>
22.             {
23.                 while (true)
24.                 {
25.                     Console.Clear();
26.                     foreach (var key in sharedList.Keys)
```

```
27.                 {
28.                     Console.WriteLine(key.ToString());
29.                 }
30.                 Thread.Sleep(1);
31.             }
32.         }).Start();
33.
34.         //Writing thread
35.         new Thread(() =>
36.         {
37.             while (true)
38.             {
39.                 Console.ReadLine();
40.                 sharedList.Add(Guid.NewGuid(), new object());
41.             }
42.         }).Start();
43.
44.     }
45. }
46.
47.
48.
```

For sure you will run into InvalidOperationException with message that collection is modified because enumeration operation is not thread safe by default.



This is because both threads are accessing shared collection at the same time, and write thread has changed the enumeration while the other one was reading. Reading state becomes invalid then and exception is thrown.

Since .Net Framework 1.1 this problem was solved by using lock. You simply use and object instance for locking and perform operation you want to make thread safe. Practically, when one thread performs lock others are waiting for lock to be released.

This way you ensure that only one thread is working with the shared collection. If locking is not involved it might cause application to work unexpectedly.

However, some collections like Hashtable provide wrappers for thread-safe operations, but this does not mean you do not need locking when enumerating through collection.

It is only for atomic operations, which enumeration is certanly not as it is executed in a loop. If you run the following code, which tries to write to Hashtable when you press enter key, you will probably get an exception. However, it might happend right in the time when reading is done, so keep pressing enter key and you'll definitely get an exception thrown.

This wrapper provides SyncRoot object for thread synchronization, but does not explicitly perform lock for enumeration. You will still have to perform lock when working with wrapped Hashtable if you are ittereating through it.

```
1.  using System;
2.  using System.Collections;
3.  using System.Collections.Concurrent;
4.  using System.Collections.Generic;
5.  using System.Threading;
6.
7.  namespace ThreadCollection
8.  {
9.      class ProgramHashtableLock
10.     {
11.         static Hashtable sharedList = Hashtable.Synchronized(new Hashtable());
12.         static void Main(string[] args)
13.         {
14.             //Initial load of 10 elements
15.             for (int i = 0; i < 10; i++)
16.             {
17.                 sharedList.Add(Guid.NewGuid(), new object());
18.             }
19.
20.             //Reading thread
21.             new Thread(() =>
22.             {
23.                 while (true)
24.                 {
```

```
25.                 Console.Clear();
26.                 lock (sharedList.SyncRoot)
27.                 {
28.                     foreach (var key in sharedList.Keys)
29.                     {
30.                         Console.WriteLine(key.ToString());
31.                     }
32.                 }
33.                 //Give some time to other threads to kick in
34.                 Thread.Sleep(1);
35.             }
36.         }).Start();
37.
38.         //Writing thread
39.         new Thread(() =>
40.         {
41.             while (true)
42.             {
43.                 Console.ReadLine();
44.                 lock (sharedList.SyncRoot)
45.                 {
46.                     sharedList.Add(Guid.NewGuid(), new object());
47.                 }
48.             }
49.         }).Start();
50.
51.     }
52. }
53. }
54.
55.
```

Starting from .Net Framework 4 completely new namespace with thread safe objects which implement self locking. My favorite is ConcurrentDictionary. It provides strongly typed thread-safe collection which can be quried, something that hashtable does not have.

```
1. using System;
2. using System.Collections;
```

```
3.  using System.Collections.Concurrent;
4.  using System.Collections.Generic;
5.  using System.Threading;
6.
7.  namespace ThreadCollection
8.  {
9.      class ProgramConcurrentDictionary
10.     {
11.         static ConcurrentDictionary<Guid,Object> sharedList = new ConcurrentDictionary<Guid,object>()
    ;
12.         static void Main(string[] args)
13.         {
14.             //Initial load of 10 elements
15.             for (int i = 0; i < 10; i++)
16.             {
17.                 sharedList.AddOrUpdate(Guid.NewGuid(), new object(), (key, oldValue) => new object());
18.             }
19.
20.             //Reading thread
21.             new Thread(() =>
22.             {
23.                 while (true)
24.                 {
25.                     Console.Clear();
26.
27.                     foreach (var key in sharedList.Keys)
28.                     {
29.                         Console.WriteLine(key.ToString());
30.                     }
31.                     //Give some time to other threads to kick in
32.                     Thread.Sleep(1);
33.                 }
34.             }).Start();
35.
36.             //Writing thread
37.             new Thread(() =>
```

```
38.             {
39.                 while (true)
40.                 {
41.                     Console.ReadLine();
42.                     sharedList.AddOrUpdate(Guid.NewGuid(), new object(), (key, oldValue) => new object());

43.                 }
44.             }).Start();
45.
46.         }
47.     }
48. }
49.
50.
```

Unless you are targetting .NET Framework 2.0 or bellow there is really no need to stick with Hashtables as is becoming obsolete in new framework versions.

Regarding other thread-safe collection from System.Collections.Concurrent namespace you can check in msdn online documentation

threadcollection.zip