

The results are in! See what nearly 90,000 developers picked as their most loved, dreaded, and desired coding languages and more in the 2019 Developer Survey.

Randomize a List<T>

[Ask Question](#)

730

What is the best way to randomize the order of a generic list in C#?
I've got a finite set of 75 numbers in a list I would like to assign a random order to, in order to draw them for a lottery type application.



c#

generic-list



216

edited May 3 '16 at 13:01



Uwe Keim

27.7k 32 135 216

asked Nov 7 '08 at 19:28



mirezus

5,228 10 29 40

- 1 There is an open issue to integrate this functionality to .NET:
github.com/dotnet/corefx/issues/461 – Natan Mar 7 '15 at 10:19
- 3 You may be interested in [this NuGet package](#), which contains extension methods for shuffling IList<T> and IEnumerable<T> using the Fisher-Yates algorithm mentioned below – ChaseMedallion May 7 '16 at 14:44

@Natan fyi, they killed it – Chris Marisic Jun 6 '16 at 15:28

There is also related [Select N random elements from a List<T>](#) and [Shuffle with OrderBy vs. Fisher-Yates](#) discussion. – Alexei Levenkov Feb 5 '17 at 7:51 ✎

Can you have an infinite set of 75 numbers ;)? – tymtam Sep 28 '17 at 3:34 ✎

18 Answers



Shuffle any `(I)List` with an extension method based on the [Fisher-Yates shuffle](#):

994



```
private static Random rng = new Random();

public static void Shuffle<T>(this IList<T> list)
{
    int n = list.Count;
    while (n > 1) {
        n--;
        int k = rng.Next(n + 1);
        T value = list[k];
        list[k] = list[n];
        list[n] = value;
    }
}
```

Usage:

```
List<Product> products = GetProducts();
products.Shuffle();
```

The code above uses the much criticised `System.Random` method to select swap candidates. It's fast but not as random as it should be. If you need a better quality of randomness in your shuffles use the random number generator in `System.Security.Cryptography` like so:

```
using System.Security.Cryptography;
...
public static void Shuffle<T>(this IList<T> list)
{
    RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider
    int n = list.Count;
    while (n > 1)
```

```

    {
        byte[] box = new byte[1];
        do provider.GetBytes(box);
        while (!(box[0] < n * (Byte.MaxValue / n)));
        int k = (box[0] % n);
        n--;
        T value = list[k];
        list[k] = list[n];
        list[n] = value;
    }
}

```

A simple comparison is available [at this blog](#) (WayBack Machine).

Edit: Since writing this answer a couple years back, many people have commented or written to me, to point out the big silly flaw in my comparison. They are of course right. There's nothing wrong with System.Random if it's used in the way it was intended. In my first example above, I instantiate the rng variable inside of the Shuffle method, which is asking for trouble if the method is going to be called repeatedly. Below is a fixed, full example based on a really useful comment received today from @weston here on SO.

Program.cs:

```

using System;
using System.Collections.Generic;
using System.Threading;

namespace SimpleLottery
{
    class Program
    {
        private static void Main(string[] args)
        {
            var numbers = new List<int>(Enumerable.Range(1, 75));
            numbers.Shuffle();
            Console.WriteLine("The winning numbers are: {0}", string.Join(
            numbers.GetRange(0, 5)));
        }
    }

    public static class ThreadSafeRandom

```

[Home](#)[PUBLIC](#)[Stack Overflow](#)[Tags](#)[Users](#)[Jobs](#)**Teams**

Q&A for work

[Learn More](#)

```

{
    [ThreadStatic] private static Random Local;

    public static Random ThisThreadsRandom
    {
        get { return Local ?? (Local = new Random(unchecked(Envirc
+ Thread.CurrentThread.ManagedThreadId))); }
    }
}

static class MyExtensions
{
    public static void Shuffle<T>(this IList<T> list)
    {
        int n = list.Count;
        while (n > 1)
        {
            n--;
            int k = ThreadSafeRandom.ThisThreadsRandom.Next(n + 1);
            T value = list[k];
            list[k] = list[n];
            list[n] = value;
        }
    }
}

```

edited Nov 14 '17 at 6:56

**Uwe Keim**

27.7k 32 135 216

answered Aug 11 '09 at 20:07

**grenade**

21.2k 17 81 118

26 What if list.Count is > Byte.MaxValue? If n = 1000, then 255 / 1000 = 0, so the do loop will be an infinite loop since box[0] < 0 is always false. – [AndrewS](#) Jun 7 '11 at 10:47

17 I would like to point out, that the comparison is flawed. Using `new Random()` in a loop is the problem, not the randomness of `Random` [Explanation](#) – [Sven](#) Sep 29 '11 at 13:43

9 It is a good idea to pass an instance of Random to the Shuffle method

rather than create it inside as if you are calling Shuffle lots of times in quick succession (e.g. shuffling lots of short lists), the lists will all be shuffled in the same way (e.g. first item always gets moved to position 3).
– [Mark Heath](#) Feb 7 '12 at 22:43

7 Just making `Random rng = new Random();` a static would solve the problem in the comparison post. As each subsequent call would follow on from the previous calls last random result. – [weston](#) Nov 28 '12 at 13:58

4 #2, it's not clear that the version with the Crypto generator works because the max range of a byte is 255, so any list larger than that will not shuffle correctly. – [Mark Sowul](#) May 8 '13 at 14:37



96



I'm bit surprised by all the clunky versions of this simple algorithm here. Fisher-Yates (or Knuth shuffle) is bit tricky but very compact. If you go to Wikipedia, you would see a version of this algorithm that has for-loop in reverse and lot of people don't really seem to understand why is it in reverse. The key reason is that this version of algorithm assumes that the random number generator `Random(n)` at your disposal has following two properties:

1. It accepts `n` as single input parameter.
2. It returns number from 0 to `n` *inclusive*.

However .Net random number generator does not satisfy #2 property. The `Random.Next(n)` instead returns number from 0 to `n-1` inclusive. If you try to use for-loop in reverse then you would need to call `Random.Next(n+1)` which adds one additional operation.

However, .Net random number generator has another nice function `Random.Next(a,b)` which returns `a` to `b-1` inclusive. This actually perfectly fits nicely with implementation of this algorithm that has normal for-loop. So without further ado, here's the correct, efficient and compact implementation:

```

public static void Shuffle<T>(this IList<T> list, Random rnd)
{
    for(var i=0; i < list.Count - 1; i++)
        list.Swap(i, rnd.Next(i, list.Count));
}

public static void Swap<T>(this IList<T> list, int i, int j)
{
    var temp = list[i];
    list[i] = list[j];
    list[j] = temp;
}

```

edited Feb 15 at 13:52



[dazbradbury](#)

4,780 5 27 35

answered Mar 26 '14 at 17:41



[Shital Shah](#)

26.1k 5 111 99

Wouldn't it be better to change `rnd(i, list.Count)` to `rnd(0, list.Count)` so that any card could be swapped? – [Donuts](#) Jul 7 '14 at 6:04

10 [@Donuts](#) - no. If you do that you will add bias in shuffle. – [Shital Shah](#) Jul 19 '14 at 7:16

2 By separating `Swap<T>` out to a separate method, seems like you cause a lot of unnecessary T allocations for temp. – [Clay](#) Dec 3 '15 at 14:52

2 I'd argue that LINQ could potentially slow the performance of the shuffling down, and that would be a reason not to use it, especially given the relative simplicity of the code. – [wanglerw28](#) Feb 12 '16 at 1:26 ✎

6 When `i = list.Count - 1`, i.e. the last iteration, `rnd.Next(i, list.Count)` will give you `i` back. You therefore need `i < list.Count - 1` as the loop condition. Well, you don't 'need' it, but it saves 1 iteration ;) – [Pod](#) May 28 '16 at 20:14 ✎

Idea is get anonymous object with item and random order and then reorder items by this order and return value:

3

```
var result = items.Select(x => new { value = x, order = rnd.Next() }
    .OrderBy(x => x.order).Select(x => x.value).ToList()
```

answered Jul 31 '18 at 6:00

[Andrey Kucher](#)

31 1

best one liner solution – [vipin8169](#) Mar 8 at 7:47

A simple modification of the [accepted answer](#) that returns a new list instead of working in-place, and accepts the more general `IEnumerable<T>` as many other Linq methods do.

0

```
private static Random rng = new Random();

/// <summary>
/// Returns a new List where the elements are randomly shuffled.
/// Based on the Fisher-Yates shuffle, which has O(n) complexity.
/// </summary>
public static IEnumerable<T> Shuffle<T>(this IEnumerable<T> list) {
    var source = list.ToList();
    int n = source.Count;
    var shuffled = new List<T>(n);
    shuffled.AddRange(source);
    while (n > 1) {
        n--;
        int k = rng.Next(n + 1);
        T value = shuffled[k];
        shuffled[k] = shuffled[n];
        shuffled[n] = value;
    }
    return shuffled;
}
```

answered Sep 8 '17 at 1:59



Extragorey

696 6 18



286

If we only need to shuffle items in a completely random order (just to mix the items in a list), I prefer this simple yet effective code that orders items by guid...



```
var shuffledcards = cards.OrderBy(a => Guid.NewGuid()).ToList();
```

edited Jan 1 '17 at 4:18



Matthew Lock

8,070 7 65 110

answered Nov 23 '10 at 23:34



user453230

3,149 1 10 2

28 GUIDs are meant to be unique not random. Part of it is machine-based and another part time-based and only a small portion is random.
blogs.msdn.com/b/oldnewthing/archive/2008/06/27/8659071.aspx – Despertar May 5 '13 at 7:00

84 This is a nice elegant solution. If you want something other than a guid to generate randomness, just order by something else. Eg:

```
var shuffledcards = cards.OrderBy(a => rng.Next());
```


compilr.com/grenade/sandbox/Program.cs – grenade May 27 '13 at 10:54

16 Please no. This is wrong. "ordering by random" is totally NOT a shuffle: you introduce a bias and, worse, you risk to go in infinite loops – Vito De Tullio Aug 16 '13 at 10:07

64 @VitoDeTullio: You are misremembering. You risk infinite loops when you provide a random *comparison function*; a comparison function is required to produce a consistent *total order*. A random *key* is fine. This suggestion

is wrong because *guids are not guaranteed to be random*, not because the technique of sorting by a random key is wrong. – [Eric Lippert](#) Sep 13 '13 at 21:30

- 21 @Doug: NewGuid only guarantees that it gives you a unique GUID. It makes no guarantees about randomness. If you're using a GUID for a purpose other than creating a *unique* value, you're doing it wrong. – [Eric Lippert](#) Sep 13 '13 at 21:31

Extension method for IEnumerable:

69

```
public static IEnumerable<T> Randomize<T>(this IEnumerable<T> source)
{
    Random rnd = new Random();
    return source.OrderBy<T, int>((item) => rnd.Next());
}
```

edited Mar 29 '16 at 10:05



[rbm](#)

2,852 2 12 26


answered Aug 11 '10 at 8:54



[Denis](#)

763 5 2

- 3 Note that this is not thread-safe, even if used on a thread-safe list – [BlueRaja - Danny Pflughoeft](#) Sep 25 '12 at 3:05
- 1 how do we give list<string> to this function ? – [MonsterMMORPG](#) Mar 7 '13 at 12:27
- 5 There are two significant problems with this algorithm: -- OrderBy uses a QuickSort variant to sort the items by their (ostensibly random) keys. QuickSort performance is $O(N \log N)$; in contrast, a Fisher-Yates shuffle is $O(N)$. For a collection of 75 elements, this may not be a big deal, but the difference will become pronounced for larger collections. – [John Beyer](#) Jun 26 '13 at 16:47

6 ... -- `Random.Next()` may produce a reasonably pseudo-random distribution of values, but it does *not* guarantee that the values will be unique. The probability of duplicate keys grows (non-linearly) with N until it reaches certainty when N reaches $2^{32}+1$. The `OrderBy QuickSort` is a *stable* sort; thus, if multiple elements happen to get assigned the same pseudo-random index value, then their order in the output sequence will be the *same* as in the input sequence; thus, a bias is introduced into the "shuffle". – [John Beyer](#) Jun 26 '13 at 17:06 

24 @JohnBeyer: There are far, far greater problems than that source of bias. There are only four billion possible seeds to `Random`, which is far, far less than the number of possible shuffles of a moderately sized set. Only a tiny fraction of the possible shuffles can be generated. That bias dwarfs the bias due to accidental collisions. – [Eric Lippert](#) Sep 13 '13 at 21:33



If you don't mind using two `Lists`, then this is probably the easiest way to do it, but probably not the most efficient or unpredictable one:

2



```
List<int> xList = new List<int>() { 1, 2, 3, 4, 5 };
List<int> deck = new List<int>();

foreach (int xInt in xList)
    deck.Insert(random.Next(0, deck.Count + 1), xInt);
```

edited Feb 21 '16 at 22:31



[Lemonseed](#)

1,107 7 23

answered Dec 22 '13 at 1:33



[Xelights](#)

69 1 3

This is my preferred method of a shuffle when it's desirable to not



modify the original. It's a variant of the [Fisher-Yates "inside-out" algorithm](#) that works on any enumerable sequence (the length of source does not need to be known from start).

```
public static IList<T> NextList<T>(this Random r, IEnumerable<T> source)
{
    var list = new List<T>();
    foreach (var item in source)
    {
        var i = r.Next(list.Count + 1);
        if (i == list.Count)
        {
            list.Add(item);
        }
        else
        {
            var temp = list[i];
            list[i] = item;
            list.Add(temp);
        }
    }
    return list;
}
```

This algorithm can also be implemented by allocating a range from 0 to length - 1 and randomly exhausting the indices by swapping the randomly chosen index with the last index until all indices have been chosen exactly once. This above code accomplishes the exact same thing but without the additional allocation. Which is pretty neat.

With regards to the `Random` class it's a general purpose number generator (and If I was running a lottery I'd consider using something different). It also relies on a time based seed value by default. A small alleviation of the problem is to seed the `Random` class with the `RNGCryptoServiceProvider` or you could use the `RNGCryptoServiceProvider` in a method similar to this (see below) to generate uniformly chosen random double floating point values but running a lottery pretty much requires understanding randomness and the nature of the randomness source.

```
var bytes = new byte[8];
_secureRng.GetBytes(bytes);
var v = BitConverter.ToInt64(bytes, 0);
return (double)v / ((double)ulong.MaxValue + 1);
```

The point of generating a random double (between 0 and 1 exclusively) is to use to scale to an integer solution. If you need to pick something from a list based on a random double x that's always going to be $0 \leq x \ \&\& \ x < 1$ is straight forward.

```
return list[(int)(x * list.Count)];
```

Enjoy!

answered Sep 19 '15 at 9:43



[John Leidegren](#)

37k 15 101 137



Old post for sure, but I just use a GUID.

-3

```
Items = Items.OrderBy(o => Guid.NewGuid().ToString()).ToList();
```



A GUID is always unique, and since it is regenerated every time the result changes each time.

edited Aug 21 '15 at 8:44



[S.L. Barth](#)

7,129 12 43 56

answered Apr 11 '15 at 16:24



[DavidMc](#)

61 1 2

Compact, but do you have a reference on the sorting of consecutive newGuids to be high quality random? Some versions of Guid/Uuid have time stamps and other non-random parts. – [Johan Lundberg](#) Dec 10 '15 at 14:47

- 7 This answer has already been given, and worse it is designed for uniqueness not randomness. – [Alex Angas](#) Jan 4 '16 at 22:21



You can achieve that by using this simple extension method

5



```
public static class IEnumerableExtensions
{
    public static IEnumerable<T> Randomize<T>(this IEnumerable<T> target)
    {
        Random r = new Random();

        return target.OrderBy(x => (r.Next()));
    }
}
```

and you can use it by doing the following

```
// use this on any collection that implements IEnumerable!
// List, Array, HashSet, Collection, etc

List<string> myList = new List<string> { "hello", "random", "world",
    "bat", "baz" };

foreach (string s in myList.Randomize())
{
    Console.WriteLine(s);
}
```

answered Aug 24 '14 at 17:48



[Shehab Fawzy](#)

3,484 1 20 17

- 2 I would keep the `Random` class instance outside the function as a `static` variable. Otherwise you might get the same randomization seed from the timer if called in quick succession. – [Lemonseed](#) Jun 2 '16 at 16:11

An interesting note - if you instantiate the `Random` class rapidly within a loop, say between 0 ms and 200 ms of each other, then you have a very high chance of getting the same randomization seed - which then results in repeating results. You can however get around this via using `Random` `rand = new Random(Guid.NewGuid().GetHashCode());` This effectively forces the randomization to be derived from the `Guid.NewGuid()` – [Baaleos](#) Feb 16 '18 at 16:28

EDIT The `RemoveAt` is a weakness in my previous version. This solution overcomes that.

8

```
public static IEnumerable<T> Shuffle<T>(
    this IEnumerable<T> source,
    Random generator = null)
{
    if (generator == null)
    {
        generator = new Random();
    }

    var elements = source.ToArray();
    for (var i = elements.Length - 1; i >= 0; i--)
    {
        var swapIndex = generator.Next(i + 1);
        yield return elements[swapIndex];
        elements[swapIndex] = elements[i];
    }
}
```

Note the optional `Random` generator, if the base framework implementation of `Random` is not thread-safe or cryptographically strong enough for your needs, you can inject your implementation into the operation.

[A suitable implementation for a thread-safe cryptographically strong Random implementation can be found in this answer.](#)

Here's an idea, extend IList in a (hopefully) efficient way:

```
public static IEnumerable<T> Shuffle<T>(this IList<T> list)
{
    var choices = Enumerable.Range(0, list.Count).ToList();
    var rng = new Random();
    for(int n = choices.Count; n > 1; n--)
    {
        int k = rng.Next(n);
        yield return list[choices[k]];
        choices.RemoveAt(k);
    }

    yield return list[choices[0]];
}
```

edited May 23 '17 at 12:18



Community ♦

1 1

answered Oct 27 '11 at 8:43



Jodrell

27k 3 59 102

See stackoverflow.com/questions/4412405/... you must be aware already. – nawfal May 30 '13 at 23:55

@nawfal see my improved implementation. – Jodrell Jul 9 '14 at 7:46

1 hmm fair enough. Is it GetNext or Next ? – nawfal Jul 9 '14 at 7:57

```
public Deck(IEnumerable<Card> initialCards)
```



```
{
    cards = new List<Card>(initialCards);
    public void Shuffle()
    {
        List<Card> NewCards = new List<Card>();
        while (cards.Count > 0)
        {
            int CardToMove = random.Next(cards.Count);
            NewCards.Add(cards[CardToMove]);
            cards.RemoveAt(CardToMove);
        }
        cards = NewCards;
    }
}

public IEnumerable<string> GetCardNames()
{
    string[] CardNames = new string[cards.Count];
    for (int i = 0; i < cards.Count; i++)
        CardNames[i] = cards[i].Name;
    return CardNames;
}

Deck deck1;
Deck deck2;
Random random = new Random();

public Form1()
{
    InitializeComponent();
    ResetDeck(1);
    ResetDeck(2);
    RedrawDeck(1);
    RedrawDeck(2);
}

private void ResetDeck(int deckNumber)
{
    if (deckNumber == 1)
    {
        int numberOfCards = random.Next(1, 11);
        deck1 = new Deck(new Card[] { });
    }
}
```



```
        for (int i = 0; i < numberOfCards; i++)
            deck1.Add(new Card((Suits)random.Next(4),(Values)random.Next(13)));
        deck1.Sort();
    }

    else
        deck2 = new Deck();
    }

    private void reset1_Click(object sender, EventArgs e) {
        ResetDeck(1);
        RedrawDeck(1);
    }

    private void shuffle1_Click(object sender, EventArgs e)
    {
        deck1.Shuffle();
        RedrawDeck(1);
    }

    private void moveToDeck1_Click(object sender, EventArgs e)
    {
        if (listBox2.SelectedIndex >= 0)
            if (deck2.Count > 0) {
                deck1.Add(deck2.Deal(listBox2.SelectedIndex));
            }

        RedrawDeck(1);
        RedrawDeck(2);
    }
}
```

answered Jun 16 '14 at 18:52




sumit laddha

21 1

- 1 Welcome to Stack Overflow! Please consider adding some explanation to your answer, rather than just a huge block of code. Our goal here is to educate people so that they understand the answer and can apply it in other situations. If you comment your code and add an explanation, you

will make your answer more helpful not just to the person who asked the question this time, but to anyone in the future who may be having the same problem. – [starsplusplus](#) Jun 16 '14 at 19:14

- 3 Most of this code is entirely irrelevant to the question, and the only useful part basically repeats Adam Tegen's answer from almost 6 years ago. – [T.C.](#) Jun 16 '14 at 19:16 

Here's a thread-safe way to do this:

0

```
public static class EnumerableExtension
{
    private static Random globalRng = new Random();

    [ThreadStatic]
    private static Random _rng;

    private static Random rng
    {
        get
        {
            if (_rng == null)
            {
                int seed;
                lock (globalRng)
                {
                    seed = globalRng.Next();
                }
                _rng = new Random(seed);
            }
            return _rng;
        }
    }

    public static IEnumerable<T> Shuffle<T>(this IEnumerable<T> item)
    {
        return items.OrderBy (i => rng.Next());
    }
}
```

answered Mar 28 '13 at 17:29



Christopher Stevenson

2,412 15 22

0

Here's an efficient Shuffler that returns a byte array of shuffled values. It never shuffles more than is needed. It can be restarted from where it previously left off. My actual implementation (not shown) is a MEF component that allows a user specified replacement shuffler.

```
public byte[] Shuffle(byte[] array, int start, int count)
{
    int n = array.Length - start;
    byte[] shuffled = new byte[count];
    for(int i = 0; i < count; i++, start++)
    {
        int k = UniformRandomGenerator.Next(n--) + start;
        shuffled[i] = array[k];
        array[k] = array[start];
        array[start] = shuffled[i];
    }
    return shuffled;
}
```

answered Jan 24 '13 at 21:26



BSalita

4,065 3 36 50

```
public static List<T> Randomize<T>(List<T> list)
{
    List<T> randomizedList = new List<T>();
}
```

10

```

Random rnd = new Random();
while (list.Count > 0)
{
    int index = rnd.Next(0, list.Count); //pick a random item
    randomizedList.Add(list[index]); //place it at the end of
    list.RemoveAt(index);
}
return randomizedList;
}

```

answered Nov 7 '08 at 21:18



Adam Tegen

14k 30 109 147

-
- 1 See stackoverflow.com/questions/4412405/... – nawfal May 30 '13 at 23:54
-
- 4 Shouldn't you do something like `var listCopy = list.ToList()` to avoid popping all of the items off the incoming list? I don't really see why you would want to mutate those lists to empty. – Chris Marisic Sep 17 '14 at 17:38
-

▲

A very simple approach to this kind of problem is to use a number of random element swap in the list.

-6

▼

In pseudo-code this would look like this:

```

do
    r1 = randomPositionInList()
    r2 = randomPositionInList()
    swap elements at index r1 and index r2
for a certain number of times

```

answered Nov 7 '08 at 19:36

Aleris



6,222 3 29 38

-
- 1 One problem with this approach is knowing when to stop. It also has a tendency to exaggerate any biases in the pseudo-random number generator. – [Mark Bessey](#) Nov 7 '08 at 19:58
-
- 3 Yes. Highly inefficient. There is no reason to use an approach like this when better, faster approaches exist that are just as simple. – [PeterAllenWebb](#) Nov 7 '08 at 21:25
-
- 1 not very efficient or effective... Running it N times would likely leave many elements in their original position. – [NSjonas](#) Dec 7 '12 at 21:46
-

▲ I usually use:

3

▼

```
var list = new List<T> ();  
fillList (list);  
var randomizedList = new List<T> ();  
var rnd = new Random ();  
while (list.Count != 0)  
{  
    var index = rnd.Next (0, list.Count);  
    randomizedList.Add (list [index]);  
    list.RemoveAt (index);  
}
```

answered Nov 7 '08 at 19:35

[albertein](#)

20.5k 4 47 57

list.RemoveAt is an O(n) operation, which makes this implementation prohibitively slow. – [George Polevoy](#) May 14 '17 at 21:09

3

If you have a fixed number (75), you could create an array with 75 elements, then enumerate your list, moving the elements to randomized positions in the array. You can generate the mapping of list number to array index using the [Fisher-Yates shuffle](#).

answered Nov 7 '08 at 19:35



dmo

2,363 4 28 38