Learn C#: Lists and LINQ

Lists in C#

In C#, a *list* is a generic data structure that can hold any type. Use the `new` operator and declare the element type in the angle brackets `<>`.

In the example code, `names` is a list containing `string` values. `someObjects` is a list containing `Object` instances.

```
List<string> names = new List<string>();
List<Object> someObjects = new
List<Object>();
```

Object Initialization

Values can be provided to a `List` when it is constructed in a process called *object initialization*. Instead of parentheses, use curly braces after the list's type.
Note that this can ONLY be used at the time of construction.

```
List<string> cities = new List<string> {
"Los Angeles", "New York City", "Dubai"
};
```

Generic Collections

Some collections, like lists and dictionaries, can be associated with various types. Instead of defining a unique class for each possible type, we define them with a generic type `T`, e.g. `List<T>`.
These collections are called *generic collection* types. They are available in the `System.Collections.Generic` namespace.
The generic type `T` will often show up in documentation. When using a generic collection in your code, the actual type is specified when the collection is declared or instantiated.

```
using System.Collections.Generic;

List<string> names = new List<string>();
List<Object> objs = new List<Object>();
Dictionary<string,int> scores = new
Dictionary<string, int>();
```

Limitless Lists

Unlike a C# array, a C# list does not have a limited number of elements. You can add as many items as you like.

```
// Initialize array with length 2
string[] citiesArray = new string[2];
citiesArray[0] = "Los Angeles";
citiesArray[1] = "New York City";
citiesArray[2] = "Dubai"; // Error!

// Initialize list; no length needed
List<string> citiesList = new
List<string>();
```

```
citiesList.Add("Los Angeles");
citiesList.Add("New York City");
citiesList.Add("Dubai");
```

Count Property

The number of elements in a list is stored in the Count property.
In the example code, the Count of citiesList changes as we add and remove values.

```
List<string> citiesList = new
List<string>();
citiesList.Add("Los Angeles");
Console.WriteLine(citiesList.Count);
// Output: 1

citiesList.Add("New York City");
Console.WriteLine(citiesList.Count);
// Output: 2

citiesList.Remove("Los Angeles");
Console.WriteLine(citiesList.Count);
// Output: 1
```

Remove()

Elements of a list can be removed with the Remove() method. The method returns true if the item is successfully removed; otherwise, false.
In the example code, attempting to remove "Cairo" returns false because that element is not in the citiesList.

```
List<string> citiesList = new
List<string>();
citiesList.Add("Los Angeles");
citiesList.Add("New York City");
citiesList.Add("Dubai");

result1 = citiesList.Remove("New York
City");
// result1 is true

result2 = citiesList.Remove("Cairo");
// result2 is false
```

Clear()

All elements of a list can be removed with the Clear() method. It returns nothing.
In the example code, the list is initialized with three items. After calling Clear(), there are zero items in the list.

```
List<string> citiesList = new
List<string> { "Delhi", "Los Angeles",
"Kiev" };
citiesList.Clear();

Console.WriteLine(citiesList.Count);
// Output: 0
```

Contains()

In C#, the list method Contains() returns true if its argument exists in the list; otherwise, false.
In the example code, the first call to Contains() returns true because "New York City" is in the list. The second call returns false because "Cairo" is not in the list.

```
List<string> citiesList = new
List<string> { "Los Angeles", "New York
City", "Dubai" };

result1 = citiesList.Contains("New York
City");
// result1 is true


result2 = citiesList.Contains("Cairo");
// result2 is false
```

List Ranges

Unlike elements in a C# array, multiple elements of a C# list can be accessed, added, or removed simultaneously. A group of multiple, sequential elements within a list is called a range.
Some common range-related methods are:
- AddRange()
- InsertRange()
- RemoveRange()

```
string[] african = new string[] {
"Cairo", "Johannesburg" };
string[] asian = new string[] { "Delhi",
"Seoul" };
List<string> citiesList = new
List<string>();

// Add two cities to the list
citiesList.AddRange(african);
// List: "Cairo", "Johannesburg"

// Add two cities to the front of the
list
citiesList.InsertRange(0, asian);
// List: "Delhi", "Seoul", "Cairo",
"Johannesburg"

// Remove the second and third cities
from the list
citiesList.RemoveRange(1, 2);
// List: "Delhi", "Johannesburg"
```

LINQ

LINQ is a set of language and framework features for writing queries on collection types. It is useful for selecting, accessing, and transforming data in a dataset.

Using LINQ

LINQ features can be used in a C# program by importing the System.Linq namespace.

```
using System.Linq;
```

var

Since the type of an executed LINQ query's result is not always known, it is common to store the result in an implicitly typed variable using the keyword var .

```
var custQuery = from cust in customers
                where cust.City ==
"Phoenix"
                select new { cust.Name,
cust.Phone };
```

Method & Query Syntax

In C#, LINQ queries can be written in *method syntax* or *query syntax*.
Method syntax resembles most other C# method calls, while query syntax resembles SQL.

```
// Method syntax
var custQuery2 = customers.Where(cust =>
cust.City == "London");

// Query syntax
var custQuery =
    from cust in customers
    where cust.City == "London"
    select cust;
```

Where

In LINQ queries, the Where operator is used to select certain elements from a sequence.
- It expects an expression that evaluates to a boolean value.
- Every element satisfying the condition will be included in the resulting query.
- It can be used in both method syntax and query syntax.

```
List<Customer> customers = new
List<Customer>
{
  new Customer("Bartleby", "London"),
  new Customer("Benjamin",
"Philadelphia"),
  new Customer("Michelle", "Busan" )
};

// Query syntax
var custQuery =
    from cust in customers
    where cust.City == "London"
    select cust;

// Method syntax
```

```
var custQuery2 = customers.Where(cust =>
cust.City == "London");


// Result: Customer("Bartleby", "London")
```

From

In LINQ queries, the from operator declares a range variable that is used to traverse the sequence. It is only used in query syntax.

In the example code, n represents each element in names. The returned query only contains those elements for which n.Contains("a") is true.

```
string[] names = { "Hansel", "Gretel",
"Helga", "Gus" };


var query =
  from n in names
  where n.Contains("a")
  select n;


// Result: Hansel, Helga
```

Select

In LINQ queries, the Select operator determines what is returned for each element in the resulting query. It can be used in both method and query syntax.

```
string[] trees = { "Elm", "Banyon",
"Rubber" };


// Query syntax
var treeQuery =
  from t in trees
  select t.ToUpper();


// Method syntax
var treeQuery2 = names.Select(t =>
t.ToUpper());


// Result: ELM, BANYON, RUBBER
```

LINQ & foreach

You can use a foreach loop to iterate over the result of an executed LINQ query.

In the example code, query is the result of a LINQ query, and it can be iterated over using foreach. name represents each element in names.

```
string[] names = { "Hansel", "Gretel",
"Helga", "Gus" };


var query = names.Where(n =>
n.Contains("a"));


foreach (var name in query)
{
  Console.WriteLine(name);
}
```

Count()

The result of an executed LINQ query has a method
Count() , which returns the number of elements it
contains.
In the example code, Count() returns 2 because the
resulting query contains 2 elements containing "a".

```
string[] names = { "Hansel", "Gretel",
"Helga", "Gus" };

var query = names.Where(x =>
x.Contains("a"));

Console.WriteLine(query.Count());
// Output: 2
```

Print   Share ▼