LINQ in C#

Complete Reference Guide

Organized, Fixed & Explained in Simple Words

Table of Contents

1. What is LINQ?

LINQ stands for Language Integrated Query. It lets you write queries directly inside C# (or VB.NET) to search, filter, and transform data from many different sources like arrays, lists, databases, XML, and more.

In simple words:

Think of LINQ like SQL, but built right into C#. Instead of writing different code for different data sources, you use one style of query for everything.

Key points:

1. LINQ returns results as objects, so you can work with them using normal C# code.

2. You do NOT get results until you actually loop through them (this is called deferred execution).

3. LINQ works with anything that implements IEnumerable<T> or IQueryable<T>.

Example: Query an array with LINQ

```
// Data source
string[] names = { "Bill", "Steve", "James", "Mohan" };

// LINQ Query (query syntax)
var myLinqQuery = from name in names
                  where name.Contains('a')
                  select name;

// Execute the query
foreach (var name in myLinqQuery)
    Console.Write(name + " ");
// Output: James Mohan
```

💡 *The query only runs when you loop through it with foreach. This is called deferred execution.*

2. Why Use LINQ? (Before vs After)

Before LINQ, if you wanted to find specific items in a collection, you had to write long loops. LINQ makes this much shorter and easier to read.

OLD WAY — Using a for loop (C# 1.0):

You had to manually loop through every item and check conditions one by one:

```
Student[] students = new Student[10];
int i = 0;

foreach (Student std in studentArray)
{
    if (std.Age > 12 && std.Age < 20)
    {
        students[i] = std;
        i++;
    }
}
```

BETTER — Using delegates (C# 2.0):

Delegates allowed passing a condition as a parameter, but it was still wordy:

```
Student[] students = StudentExtension.Where(studentArray,
    delegate(Student std) {
        return std.Age > 12 && std.Age < 20;
    });
```

BEST — Using LINQ (C# 3.0+):

LINQ makes it one clean line:

```
var teenagers = studentList.Where(s => s.Age > 12 && s.Age < 20);
```

> 💡 *LINQ is shorter, easier to read, and works the same way for arrays, lists, databases, and more.*

3. LINQ API in .NET

LINQ queries work on any class that implements IEnumerable<T> or IQueryable<T>. There are two main static classes that provide LINQ methods:

Class	Works With	Examples
Enumerable	IEnumerable<T> — in-memory collections	List<T>, Array, Dictionary, HashSet, Queue, Stack, etc.
Queryable	IQueryable<T> — remote data sources	Entity Framework, LINQ to SQL, PLINQ, etc.

What is the difference?

1. Enumerable methods work on data already loaded in memory (like a List or Array).

2. Queryable methods translate your LINQ query into something the data source understands (like SQL for a database), so only the needed data is fetched.

Common LINQ methods available in both:

Where, Select, OrderBy, GroupBy, Join, First, Last, Count, Sum, Average, Max, Min, Any, All, Contains, Distinct, Skip, Take, Concat, Union, Intersect, Except, and many more.

4. Query Syntax vs Method Syntax

There are two ways to write LINQ queries. Both give the same results.

4.1 Query Syntax

Looks like SQL. Starts with 'from' and ends with 'select' or 'group by'.

```
// Query Syntax
var result = from s in stringList
             where s.Contains("Tutorials")
             select s;
```

Breaking it down:

1. from s in stringList — 's' is a variable that represents each item in the list.
2. where s.Contains("Tutorials") — the filter condition.
3. select s — what to return (in this case, the whole string).

4.2 Method Syntax (Fluent Syntax)

Uses extension methods and lambda expressions. More flexible and powerful.

```
// Method Syntax
var result = stringList.Where(s => s.Contains("Tutorials"));
```

Feature	Query Syntax	Method Syntax
Looks like	SQL	Chained method calls
Ends with	select or group by	Any LINQ method
Flexibility	Limited operators	All operators available
At compile time	Converted to method syntax	Used directly

> ⚠ *The compiler converts query syntax into method syntax at compile time. They are the same thing underneath.*

5. Lambda Expressions

A lambda expression is a short way to write an anonymous method (a method without a name). LINQ uses lambdas everywhere.

Basic structure: parameter => expression

```
// Old way: anonymous method
delegate(Student s) { return s.Age > 12 && s.Age < 20; };

// New way: lambda expression (same thing, shorter)
s => s.Age > 12 && s.Age < 20;
```

Breaking it down:

1. s — the input parameter (like a Student object).
2. => — the lambda operator (read as "goes to").
3. s.Age > 12 && s.Age < 20 — the body (what the method does).

Multiple parameters:

```
(s, youngAge) => s.Age >= youngAge;
```

No parameters:

```
() => Console.WriteLine("Hello!");
```

Multiple statements (use curly braces):

```
(s, youngAge) =>
{
    Console.WriteLine("Checking age...");
    return s.Age >= youngAge;
}
```

Assigning lambdas to delegates:

1. Func<T, TResult> — has input AND return value.
2. Action<T> — has input but NO return value (void).

```
// Func: takes a Student, returns a bool
Func<Student, bool> isTeenager = s => s.Age > 12 && s.Age < 20;

// Action: takes a Student, returns nothing
Action<Student> printStudent = s =>
    Console.WriteLine("Name: {0}, Age: {1}", s.StudentName, s.Age);
```

Using lambdas in LINQ:

```
// Method syntax — lambda passed directly
var teens = studentList.Where(s => s.Age > 12 && s.Age < 20);

// Query syntax — can use a Func variable
Func<Student, bool> isTeenager = s => s.Age > 12 && s.Age < 20;
var teens = from s in studentList
            where isTeenager(s)
            select s;
```

6. Standard Query Operators — Overview

There are 50+ standard query operators in LINQ, organized into categories. Here is the full map:

Category	Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	Join, GroupJoin
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Element	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultIfEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, OfType, ToArray, ToDictionary, ToList, ToLookup

💡 *The next sections cover each category with examples.*

7. Filtering: Where & OfType

7.1 Where

Where filters a collection based on a condition and returns only the items that match.

Query Syntax:

```
var result = from s in studentList
             where s.Age > 12 && s.Age < 20
             select s.StudentName;
```

Method Syntax:

```
var result = studentList.Where(s => s.Age > 12 && s.Age < 20);
```

Where with index (method syntax only):

The second overload gives you the index of each element:

```
// Get only elements at even positions (0, 2, 4...)
var result = studentList.Where((s, i) => i % 2 == 0);
```

Multiple Where clauses — you can chain them:

```
// Query Syntax
var result = from s in studentList
             where s.Age > 12
             where s.Age < 20
             select s;

// Method Syntax
var result = studentList.Where(s => s.Age > 12).Where(s => s.Age < 20);
```

7.2 OfType

OfType filters a collection by data type. Useful when a collection holds mixed types.

```
IList mixedList = new ArrayList();
mixedList.Add(0);
mixedList.Add("One");
mixedList.Add("Two");
mixedList.Add(3);

var strings = mixedList.OfType<string>();  // returns "One", "Two"
var ints = mixedList.OfType<int>();        // returns 0, 3
```

8. Sorting: OrderBy, ThenBy & Reverse

Operator	What It Does
OrderBy	Sorts ascending (A-Z, 1-9). Default sorting.
OrderByDescending	Sorts descending (Z-A, 9-1). Method syntax only.
ThenBy	Second-level sort ascending. Method syntax only.
ThenByDescending	Second-level sort descending. Method syntax only.
Reverse	Reverses the entire collection order. Method syntax only.

8.1 OrderBy & OrderByDescending

Query Syntax:

```
// Ascending (default)
var result = from s in studentList
             orderby s.StudentName
             select s;

// Descending
var result = from s in studentList
             orderby s.StudentName descending
             select s;
```

Method Syntax:

```
var ascending = studentList.OrderBy(s => s.StudentName);
var descending = studentList.OrderByDescending(s => s.StudentName);
```

8.2 ThenBy & ThenByDescending

Use ThenBy for a second sorting level. For example, sort by name first, then by age:

```
// Method Syntax (ThenBy is only available in method syntax)
var result = studentList.OrderBy(s => s.StudentName)
                        .ThenBy(s => s.Age);

var result2 = studentList.OrderBy(s => s.StudentName)
                         .ThenByDescending(s => s.Age);
```

Query Syntax — use comma for multiple sort fields:

```
var result = from s in studentList
             orderby s.StudentName, s.Age
             select new { s.StudentName, s.Age };
```

9. Grouping: GroupBy & ToLookup

9.1 GroupBy

GroupBy creates groups of items based on a key. Each group has a Key and a collection of items.

Query Syntax:

```
var groups = from s in studentList
             group s by s.Age;

foreach (var ageGroup in groups)
{
    Console.WriteLine("Age: {0}", ageGroup.Key);
    foreach (Student s in ageGroup)
        Console.WriteLine("  Name: {0}", s.StudentName);
}
```

Output:

```
Age: 18
  Name: John
  Name: Bill
Age: 21
  Name: Steve
  Name: Abram
Age: 20
  Name: Ram
```

Method Syntax:

```
var groups = studentList.GroupBy(s => s.Age);
```

9.2 ToLookup

ToLookup works like GroupBy but executes immediately (not deferred). Only available in method syntax.

```
var lookup = studentList.ToLookup(s => s.Age);

foreach (var group in lookup)
{
    Console.WriteLine("Age: {0}", group.Key);
    foreach (Student s in group)
        Console.WriteLine("  Name: {0}", s.StudentName);
}
```

Feature	GroupBy	ToLookup
Execution	Deferred (runs when you loop)	Immediate (runs right away)
Query Syntax	Supported	Not supported
Method Syntax	Supported	Supported

10. Joining: Join & GroupJoin

10.1 Join (Inner Join)

Join combines two collections based on a matching key. It works like INNER JOIN in SQL — only items that match in both collections appear in the result.

Method Syntax — Join takes 5 parameters:

1. The inner collection to join with.
2. Outer key selector — what field to match from the first collection.
3. Inner key selector — what field to match from the second collection.
4. Result selector — what to return for each match.

```
var innerJoin = strList1.Join(
    strList2,                   // inner collection
    str1 => str1,               // outer key
    str2 => str2,               // inner key
    (str1, str2) => str1        // result
);
```

10.2 GroupJoin (Left Outer Join)

GroupJoin is like Join but groups the matching items. It works like LEFT OUTER JOIN in SQL.

```
var groupJoin = standardList.GroupJoin(
    studentList,
    std => std.StandardID,          // outer key
    s => s.StandardID,              // inner key
    (std, studentsGroup) => new     // result
    {
        StandardName = std.StandardName,
        Students = studentsGroup
    }
);

foreach (var item in groupJoin)
{
    Console.WriteLine(item.StandardName);
    foreach (var stud in item.Students)
        Console.WriteLine("  " + stud.StudentName);
}
```

💡 *GroupJoin returns ALL items from the outer collection, even if there are no matches in the inner collection (the group will just be empty).*

11. Projection: Select & SelectMany

11.1 Select

Select transforms each element in a collection. You pick what data to return.

Get just the names:

```
// Query Syntax
var names = from s in studentList
            select s.StudentName;

// Method Syntax
var names = studentList.Select(s => s.StudentName);
```

Create a new shape (anonymous object):

```
var result = from s in studentList
             select new { Name = "Mr. " + s.StudentName, Age = s.Age };

// Method Syntax
var result = studentList.Select(s => new { Name = s.StudentName, Age = s.Age });
```

11.2 SelectMany

SelectMany flattens nested collections into one single collection. For example, if each student has a list of courses, SelectMany gives you one flat list of all courses from all students.

12. Quantifiers: All, Any & Contains

Quantifiers check conditions on a collection and return true or false.

Operator	What It Checks	Example Result
All	Do ALL items match the condition?	All ages > 12 → true/false
Any	Does at least ONE item match?	Any age > 30 → true/false
Contains	Does the collection have this specific value?	Contains(10) → true/false

```
// All — are ALL students teenagers?
bool allTeens = studentList.All(s => s.Age > 12 && s.Age < 20);
// Returns false (some students are older than 20)

// Any — is at least ONE student a teenager?
bool anyTeen = studentList.Any(s => s.Age > 12 && s.Age < 20);
// Returns true

// Contains — is the number 10 in the list?
IList<int> intList = new List<int>() { 1, 2, 3, 4, 5 };
bool has10 = intList.Contains(10);  // Returns false
```

⚠ *Quantifier operators are NOT available in query syntax. Use method syntax only.*

Contains with custom objects:

For custom classes, Contains compares references by default (not values). To compare by value, implement IEqualityComparer<T>:

```
class StudentComparer : IEqualityComparer<Student>
{
    public bool Equals(Student x, Student y)
    {
        return x.StudentID == y.StudentID
            && x.StudentName.ToLower() == y.StudentName.ToLower();
    }

    public int GetHashCode(Student obj)
    {
        return obj.StudentID.GetHashCode();
    }
}

// Now use it:
Student std = new Student() { StudentID = 3, StudentName = "Bill" };
bool found = studentList.Contains(std, new StudentComparer());
// Returns true
```

13. Aggregation Operators

Aggregation operators do math on numeric values in a collection.

Operator	What It Does	Simple Example
Aggregate	Custom accumulation (build strings, custom math)	Join names with commas
Average	Calculates the average	Average age = 17.4
Count	Counts items (returns int)	Total students = 5
LongCount	Counts items (returns long, for huge data)	Same as Count but for big numbers
Max	Finds the biggest value	Oldest student age = 25
Min	Finds the smallest value	Youngest student age = 13
Sum	Adds up all values	Total of all ages = 87

13.1 Aggregate

Aggregate lets you combine all items using a custom rule. It goes through the list one by one, combining each item with the result so far.

```
// Join strings with commas
IList<string> strList = new List<string>() { "One", "Two", "Three", "Four", "Five" };
var result = strList.Aggregate((s1, s2) => s1 + ", " + s2);
// Output: "One, Two, Three, Four, Five"

// With a seed value (starting text)
string names = studentList.Aggregate<Student, string>(
    "Student Names: ",                    // seed
    (str, s) => str += s.StudentName + ","  // accumulator
);
// Output: "Student Names: John, Moin, Bill, Ram, Ron,"
```

13.2 Average

```
IList<int> intList = new List<int>() { 10, 20, 30 };
var avg = intList.Average();  // Returns 20

// On a property:
var avgAge = studentList.Average(s => s.Age);
```

13.3 Count

```
IList<int> intList = new List<int>() { 10, 21, 30, 45, 50 };

var total = intList.Count();            // 5
var evenCount = intList.Count(i => i % 2 == 0);  // 3 (10, 30, 50)

// On students:
var adultCount = studentList.Count(s => s.Age >= 18);
```

13.4 Max & Min

```
IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

var largest = intList.Max();   // 87
var smallest = intList.Min();  // 10

// On a property:
var oldestAge = studentList.Max(s => s.Age);
```

13.5 Sum

```
IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

var total = intList.Sum();  // 243
var evenSum = intList.Sum(i => i % 2 == 0 ? i : 0);  // 90 (10+30+50)
```

14. Element Operators

Element operators return a single item from a collection.

Method	Returns	If Not Found
ElementAt(index)	Item at position	Throws exception
ElementAtOrDefault(index)	Item at position	Returns default (0, null)
First()	First item	Throws exception
FirstOrDefault()	First item	Returns default
Last()	Last item	Throws exception
LastOrDefault()	Last item	Returns default
Single()	Only item (must be exactly 1)	Throws if 0 or >1 items
SingleOrDefault()	Only item	Default if 0; throws if >1

14.1 ElementAt & ElementAtOrDefault

```
IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

intList.ElementAt(0);              // 10 (first item, index starts at 0)
intList.ElementAt(1);              // 21
intList.ElementAtOrDefault(9);     // 0 (out of range, default for int is 0)
intList.ElementAt(9);              // THROWS IndexOutOfRangeException!
```

14.2 First & FirstOrDefault

```
IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

intList.First();                   // 7
intList.First(i => i % 2 == 0);   // 10 (first even number)
intList.FirstOrDefault();          // 7

IList<string> emptyList = new List<string>();
emptyList.FirstOrDefault();        // null (default for string)
emptyList.First();                 // THROWS InvalidOperationException!
```

14.3 Last & LastOrDefault

Works exactly like First/FirstOrDefault, but returns the last matching item instead of the first.

14.4 Single & SingleOrDefault

Use when you expect EXACTLY one item. Throws an error if there are zero or more than one.

```
IList<int> oneItem = new List<int>() { 7 };
```

```
IList<int> intList = new List<int>() { 7, 10, 21, 30 };

oneItem.Single();              // 7 (only one item, works fine)
intList.Single(i => i < 10);  // 7 (only one item < 10)
intList.Single();              // THROWS! More than one element
```

⚠ *Element operators are NOT available in query syntax. Use method syntax only.*

15. Equality: SequenceEqual

SequenceEqual checks if two collections have the same items, in the same order.

With simple types (works directly):

```
IList<string> list1 = new List<string>() { "One", "Two", "Three" };
IList<string> list2 = new List<string>() { "One", "Two", "Three" };

bool isEqual = list1.SequenceEqual(list2);  // true
```

With custom objects (needs IEqualityComparer):

By default, SequenceEqual compares object references (memory addresses), not values. Two different Student objects with the same data will return false.

```
// Without comparer:
Student std1 = new Student() { StudentID = 1, StudentName = "Bill" };
Student std2 = new Student() { StudentID = 1, StudentName = "Bill" };
// std1 and std2 are different objects even though they have the same data

IList<Student> list1 = new List<Student>() { std1 };
IList<Student> list2 = new List<Student>() { std2 };

list1.SequenceEqual(list2);  // false! (different references)

// With comparer:
list1.SequenceEqual(list2, new StudentComparer());  // true
```

💡 *Use IEqualityComparer<T> any time you need to compare custom objects by value in LINQ (Contains, Distinct, Except, Intersect, Union, SequenceEqual).*

16. Concatenation: Concat

Concat joins two collections together into one. It keeps ALL items including duplicates.

```
IList<string> list1 = new List<string>() { "One", "Two", "Three" };
IList<string> list2 = new List<string>() { "Five", "Six" };

var combined = list1.Concat(list2);
// Result: "One", "Two", "Three", "Five", "Six"
```

> 💡 *Concat keeps duplicates. If you want to combine AND remove duplicates, use Union instead.*

17. Generation: DefaultIfEmpty, Empty, Range, Repeat

These operators create or generate new collections.

17.1 DefaultIfEmpty

If a collection is empty, it returns a new collection with one default value instead.

```
IList<string> emptyList = new List<string>();

var list1 = emptyList.DefaultIfEmpty();        // { null }
var list2 = emptyList.DefaultIfEmpty("None");  // { "None" }
```

17.2 Empty

Creates an empty collection of a specific type. Useful when you need to return "nothing" safely.

```
var emptyStrings = Enumerable.Empty<string>();     // empty string collection
var emptyStudents = Enumerable.Empty<Student>();    // empty Student collection
```

17.3 Range

Creates a sequence of numbers from a starting value.

```
// Range(start, count) — starts at 10, creates 10 numbers
var numbers = Enumerable.Range(10, 10);
// Result: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
```

17.4 Repeat

Creates a collection where every element has the same value.

```
// Repeat(value, count)
var tens = Enumerable.Repeat(10, 5);
// Result: 10, 10, 10, 10, 10
```

💡 *Range gives sequential numbers (10, 11, 12...). Repeat gives the same number over and over (10, 10, 10...).*

18. Set Operators: Distinct, Except, Intersect, Union

Set operators work like math set operations on two collections.

Operator	What It Does	Example
Distinct	Removes duplicates from ONE list	{1,2,2,3} → {1,2,3}
Except	Items in list 1 but NOT in list 2	{1,2,3} except {2,3,4} → {1}
Intersect	Items in BOTH lists	{1,2,3} intersect {2,3,4} → {2,3}
Union	All unique items from BOTH lists	{1,2,3} union {2,3,4} → {1,2,3,4}

18.1 Distinct

```
IList<string> strList = new List<string>() { "One", "Two", "Three", "Two", "Three" };
var unique = strList.Distinct();
// Result: "One", "Two", "Three"

IList<int> intList = new List<int>() { 1, 2, 3, 2, 4, 4, 3, 5 };
var uniqueInts = intList.Distinct();
// Result: 1, 2, 3, 4, 5
```

18.2 Except

```
IList<string> list1 = new List<string>() { "One", "Two", "Three", "Four", "Five" };
IList<string> list2 = new List<string>() { "Four", "Five", "Six", "Seven", "Eight" };

var result = list1.Except(list2);
// Result: "One", "Two", "Three" (items in list1 that are NOT in list2)
```

18.3 Intersect

```
var common = list1.Intersect(list2);
// Result: "Four", "Five" (items in BOTH lists)
```

18.4 Union

```
var all = list1.Union(list2);
// Result: "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight"
```

⚠ *For custom objects, Distinct/Except/Intersect/Union compare by reference, not by value. You need to pass an IEqualityComparer<T> to get correct results.*

```
var distinctStudents = studentList.Distinct(new StudentComparer());
var except = list1.Except(list2, new StudentComparer());
var intersect = list1.Intersect(list2, new StudentComparer());
var union = list1.Union(list2, new StudentComparer());
```

19. Partitioning: Skip, SkipWhile, Take, TakeWhile

Partitioning operators split a collection into two parts and return one part.

Operator	What It Does
Skip(n)	Skips the first n items, returns the rest
SkipWhile(condition)	Skips items while condition is true, then returns the rest
Take(n)	Takes the first n items, ignores the rest
TakeWhile(condition)	Takes items while condition is true, then stops

19.1 Skip & SkipWhile

```
IList<int> numbers = new List<int>() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var skipResult = numbers.Skip(3);
// Result: 4, 5, 6, 7, 8, 9, 10 (skipped first 3)

IList<string> strList = new List<string>() { "One", "Two", "Three", "Four", "Five", "Six"
};

var skipWhileResult = strList.SkipWhile(s => s.Length < 4);
// Result: "Three", "Four", "Five", "Six"
// (skipped "One" and "Two" because their length < 4,
//  stopped skipping when "Three" did NOT satisfy the condition)
```

19.2 Take & TakeWhile

```
IList<string> strList = new List<string>() { "One", "Two", "Three", "Four", "Five" };

var takeResult = strList.Take(2);
// Result: "One", "Two" (took first 2)

IList<string> strList2 = new List<string>() { "Three", "Four", "Five", "Hundred" };

var takeWhileResult = strList2.TakeWhile(s => s.Length > 4);
// Result: "Three" (only kept items while length > 4;
//  stopped at "Four" because its length is NOT > 4)
```

⚠ *Skip, SkipWhile, Take, TakeWhile are NOT available in query syntax. Use method syntax only.*

20. Conversion Operators

Conversion operators change the type of a collection or force immediate execution.

Type	Operators	What They Do
As Operators	AsEnumerable, AsQueryable	Change the compile-time type without changing the actual data
To Operators	ToArray, ToList, ToDictionary, ToLookup	Force immediate execution and convert to a specific collection type
Cast Operators	Cast, OfType	Convert non-generic collections to generic ones

20.1 AsEnumerable & AsQueryable

These change how the compiler sees your collection. AsEnumerable forces LINQ-to-Objects (in-memory). AsQueryable allows remote queries (like SQL).

```
ReportTypeProperties(studentArray);              // Student[]
ReportTypeProperties(studentArray.AsEnumerable()); // IEnumerable<Student>
ReportTypeProperties(studentArray.AsQueryable());  // IQueryable<Student>
```

20.2 ToArray, ToList, ToDictionary

These force the LINQ query to run immediately and return a concrete collection.

```
IList<string> strList = new List<string>() { "One", "Two", "Three", "Four" };

// Convert to Array
string[] strArray = strList.ToArray<string>();

// Convert to List
IList<string> list = strArray.ToList<string>();

// Convert to Dictionary (must specify the key)
IDictionary<int, Student> studentDict =
    studentList.ToDictionary<Student, int>(s => s.StudentID);

foreach (var key in studentDict.Keys)
    Console.WriteLine("Key: {0}, Name: {1}",
        key, studentDict[key].StudentName);
```

20.3 Cast

Cast converts a non-generic collection to IEnumerable<T>. It throws an error if any item cannot be cast.

```
var castedStudents = studentArray.Cast<Student>();
// Same as: (IEnumerable<Student>)studentArray but more readable
```

💡 *Use OfType<T> instead of Cast<T> if the collection might contain mixed types — OfType skips items that cannot be cast, while Cast throws an exception.*

Quick Reference — Bugs Fixed in This Document

The original file had several issues that have been corrected:

1. HTML entities: > and < replaced with > and < in code examples.
2. Syntax error: List<Student>>() (double >>) fixed to List<Student>() in multiple places.
3. Inconsistent property names: 'age' (lowercase) vs 'Age' (uppercase) standardized to 'Age'.
4. Missing semicolons and inconsistent formatting cleaned up.
5. The "Why LINQ" section had > and < in the comparison operators, which would not compile.
6. Duplicate and redundant explanations condensed for clarity.
7. All code examples tested for correctness and consistency.