Generic Collections and LINQ

Contents

Introduction to collection objects and LINQ

- .NET provides a set of pre-packaged data structures known as collections
  - They have been carefully designed to ensure robust and efficient performance
  - *Array*
    - Efficient random access but inefficient to resize it
  - *List*
    - Enables dynamic resizing but not random access

Introduction to collection objects and LINQ

- In addition, C# provides a mechanism for querying collections known as LINQ
  - *Language Integrated Query*
- LINQ enables access to collections (and databases!) using *query expressions* which are similar to SQL queries
  - This allows the retrieval of information from a wide variety of data sources

Introduction to collection objects and LINQ

- We will primarily look at *LINQ to Objects*
  - Enables the querying on collection objects
- .NET also provides LINQ providers for :
  - *LINQ to SQL*
    - For querying databases
  - *LINQ to XML*
    - For querying xml documents

Querying an array

- We can design a simple LINQ query which filters the contents of an array
  - Applying the query to the array causes all the values of the array which meet a certain criteria to be extracted
  - But the query doesn't say *how* the iteration through the array is performed
    - All the necessary code is generated by the compiler, the query just specifies the criteria

Querying an array

- A simple query object comprises *from, where* and *select* clauses
  - Also, we can make use of the keyword *var* which is an implicit type
    - Essentially a strongly typed local variable but the type is determined by the compiler
    - In this case the type of the array is determined by the data source type

```
var filteredArray =
     from ......    // range variable and data source
     where .....    // boolean expression
     select.....    // which value appears in the results
```

```
using System;
using System.Collections.Generic;
using System.Linq;

class LINQtoArray
{
      static void Main(string[] args)
      {
            int[] array = { 2, 6, 4, 12, 7, 8, 9, 13, 2 };

            var filteredArray =                 // LINQ query
                  from element in array
                  where element < 7
                  select element;

            PrintArray(filteredArray, "All values less than 7:");
      }

      public static void PrintArray(IEnumerable<int> arr, string message)
      {
            Console.Write("{0}",message);

            foreach (var element in arr)
                  Console.Write(" {0}", element);

            Console.WriteLine();
      }
}
```

Querying an array

- *IEnumerable<T>* is an interface implemented by arrays and collections
- It is a *generic type*
  - We replace the *T* by a real type (such as an *int*)
  - More on this later

Querying an array

- We can add the *orderby* (descending) clause to our query to sort our filtered array into ascending (descending) order

```
var filteredArray =
      from ......                    // range variable and data source
      where .....                    // boolean expression
      orderby ..... (descending)     // sort
      select.....                    // which value appears in the results
```

```
using System;
using System.Collections.Generic;
using System.Linq;

class LINQtoArray
{
      static void Main(string[] args)
      {
            int[] array = { 2, 6, 4, 12, 7, 8, 9, 13, 2 };

            var filteredArray =                 // LINQ query
                  from element in array
                  where element < 7
                  select element;

            PrintArray(filteredArray, "All values less than 7:");

            var orderedFilteredArray =
                  from element in filteredArray
                  orderby element
                  select element;

            PrintArray(orderedFilteredArray, "All values less than 7 and sorted:");
      }

      public static void PrintArray(IEnumerable<int> arr, string message)
      {……}
}
```

Querying an array

- It's important to understand a feature of LINQ known as *deferred execution*
  - The result of a LINQ query expression **is not** a sequence or collection of objects but a **query object**
  - It represents the commands needed to execute the query
  - The query *does not execute until the program requests data from the query object*
  - Deferred execution is a powerful feature of LINQ as it allows applications to pass queries around as data
    - In our simple example, the query is not run until it is passed to the *PrintArray* method

Querying an array

- We can use LINQ to query an array of user defined objects or strings
- We must be careful when using *orderby*
  - The objects must be comparable
  - Comparable types in .NET implement the *IComparable<T>* interface
  - Built in primitive types automatically implement *IComparable<T>*
    - The '*T*' is a parameterised type
    - We will look at these in more detail later
- For example we can query an array of *StudentInfo* objects

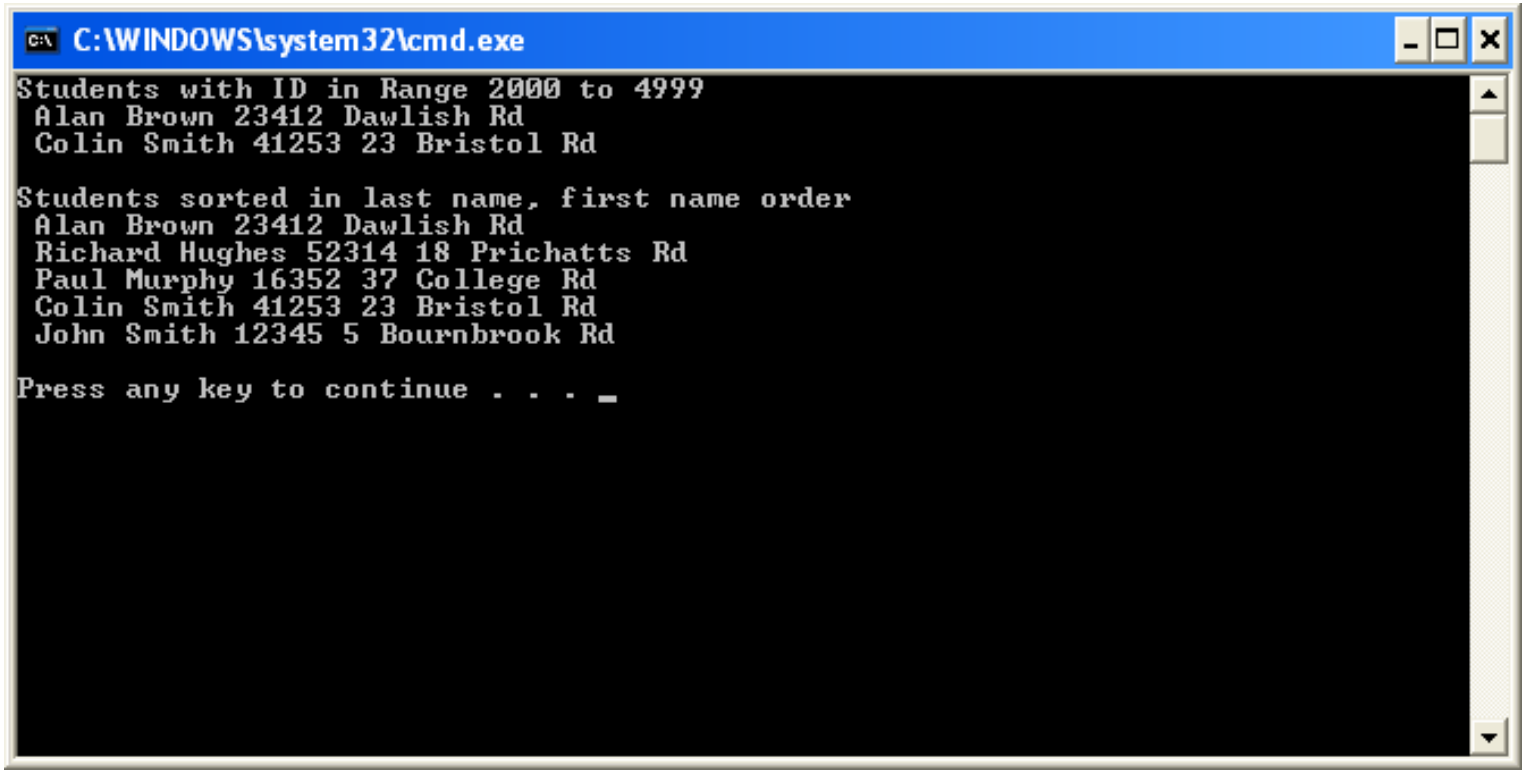
```
class StudentInfo
{
	public StudentInfo(string ln, string fn, int id, string a)
	{
		lastName = ln; firstName = fn; idNumber = id; address = a;
	}

	public override string ToString()
	{
		return firstName+" "+lastName+" "+idNumber+" " +address;
	}

	public string FirstName
	{
		get { return firstName; }
	}

	public string LastName
	{
		get { return lastName; }
	 }

	public string Address
	{
		get { return address; }
	}

	public int ID
	{
		get { return idNumber; }
	}

	private string firstName,lastName;
	private int idNumber;
	private string address;
}
```

Querying an array

- We can filter the array by ID number
  - Simply get the ID property of the range variable
- Also we can sort the names into last name order and then first name order using *orderby*
  - This uses the fact that the *string* type implements *IComparable<T>*

```
public class LINQtoObjectArray
{
      static void Main(string[] args)
      {
            StudentInfo[] students ={
            new StudentInfo("Smith", "John", 12345, "5 Bournbrook Rd"),
            new StudentInfo("Brown", "Alan", 23412, "Dawlish Rd"),
            new StudentInfo("Smith","Colin", 41253, "23 Bristol Rd"),
            new StudentInfo("Hughes", "Richard", 52314, "18 Prichatts Rd"),
            new StudentInfo("Murphy", "Paul", 16352, "37 College Rd") };

            // Filter a range of ID numbers
            var idRange=
                  from s in students
                  where s.ID>19999 && s.ID<=49999
                  select s;

            PrintArray(idRange,"Students with ID in Range 2000 to 4999");

            // Order by last name and then first name
            var nameSorted =
                  from s in students
                  orderby s.LastName, s.FirstName
                  select s;

            PrintArray(nameSorted, "Students sorted in last name, first name
                        order");
      }

      public static void PrintArray<T>(IEnumerable<T> arr, string message)
      {…}
}
```

Querying an array



Querying an array

- We can use LINQ to sort the array of *StudentInfo* objects by implementing the *IComparable* interface
  - We simply need to implement the *CompareTo()* method
  - If this isn't done when we try and sort an array of objects using LINQ, a runtime exception is generated

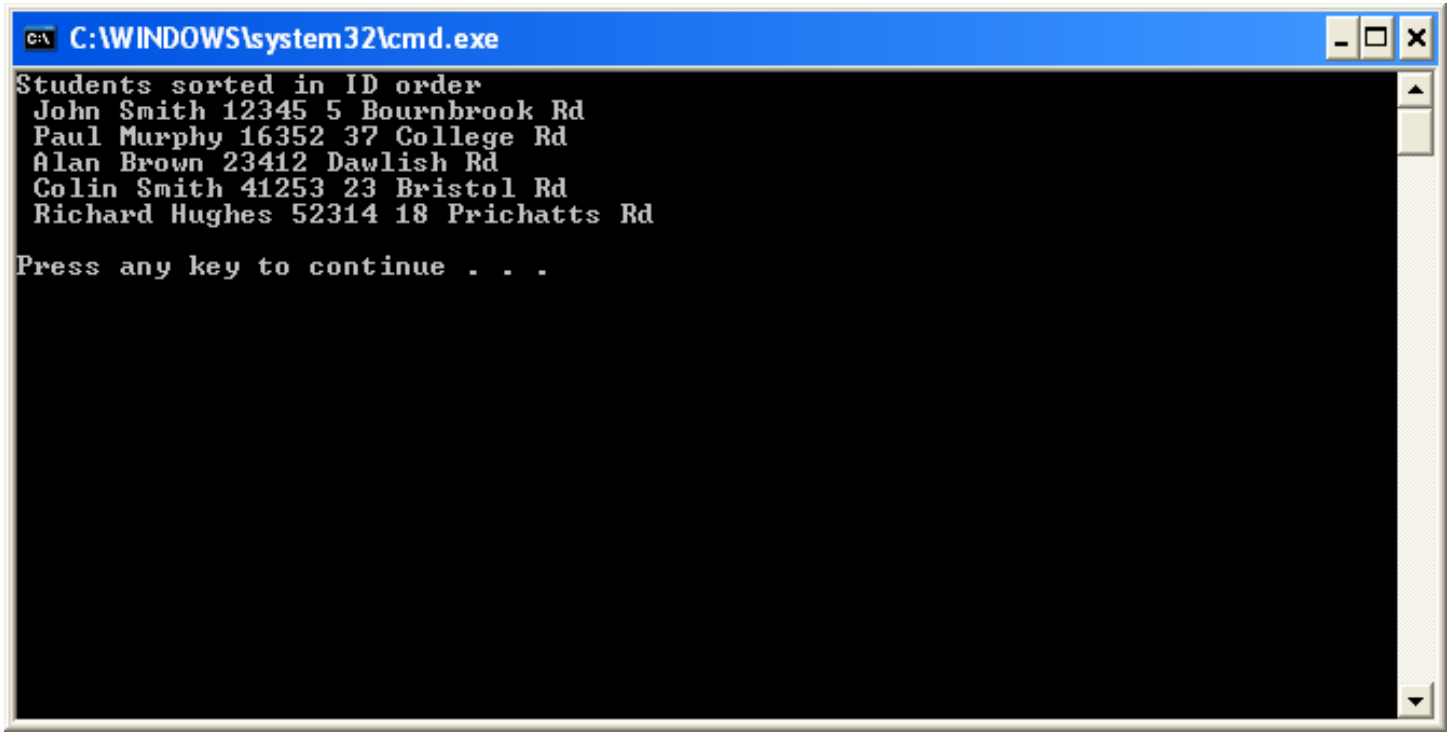
Querying an array

```
class StudentInfo : IComparable
{
	public StudentInfo(string ln, string fn, int id, string a)
	{
		lastName = ln; firstName = fn; idNumber = id; address = a;
	}

	public int CompareTo(object obj)
	{
		StudentInfo s = (StudentInfo)obj;

		if (s.ID < ID)
			return 1;
		else if (s.ID > ID)
			return -1;
		else
			return 0;
	}

	.
	.
	.
	.

	private string firstName,lastName;
	private int idNumber;
	private string address;
}
```

Querying an array

```
public class LINQtoObjectArray
{
      static void Main(string[] args)
      {
            StudentInfo[] students ={
                  new StudentInfo("Smith", "John", 12345, "5 Bournbrook Rd"),
                  new StudentInfo("Brown", "Alan", 23412, "Dawlish Rd"),
                  new StudentInfo("Smith","Colin", 41253, "23 Bristol Rd"),
                  new StudentInfo("Hughes", "Richard", 52314, "18 Prichatts Rd"),
                  new StudentInfo("Murphy", "Paul", 16352, "37 College Rd") };


            // Order by ID
            var IDSorted =
                  from s in students
                  orderby s
                  select s;

            PrintArray(IDSorted, "Students sorted in ID order");


            public static void PrintArray<T>(IEnumerable<T> arr, string message)
            {…}
      }
}
```

Querying an array

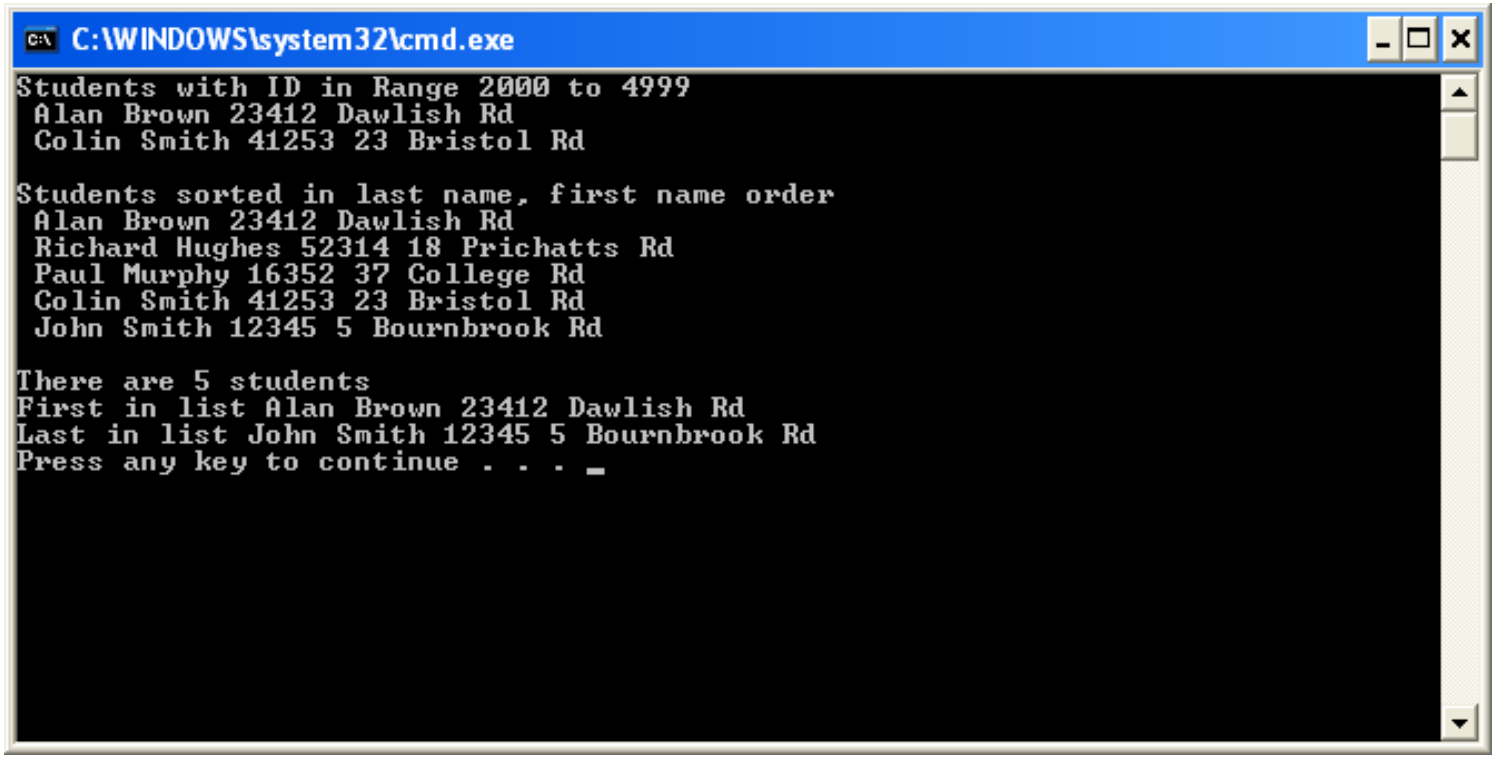


Querying an array

- LINQ defines a number of extension methods of *IEnumerable<T>*
  - Extension methods extend the functionality of *existing* classes (including classes in the FCL)
    - *Any()*
      - Checks to see if the container has any members
    - *Count()*
      - Returns a count of the number of members
    - *First(), Last()*
      - Returns the first and last members of the container
    - *Distinct()*
      - Removes duplicate members

```
public class LINQtoObjectArray
{
      static void Main(string[] args)
      {
            StudentInfo[] students ={
            new StudentInfo("Smith", "John", 12345, "5 Bournbrook Rd"),
            new StudentInfo("Brown", "Alan", 23412, "Dawlish Rd"),
            new StudentInfo("Smith","Colin", 41253, "23 Bristol Rd"),
            new StudentInfo("Hughes", "Richard", 52314, "18 Prichatts Rd"),
            new StudentInfo("Murphy", "Paul", 16352, "37 College Rd") };


            // Order by last name and then first name
             var nameSorted =
                  from s in students
                  orderby s.LastName, s.FirstName
                  select s;

            PrintArray(nameSorted, "Students sorted in last name, first name
                  order");

            Console.WriteLine("There are " + students.Count() + " students");

            if (nameSorted.Any())
            {
                  Console.WriteLine("First in list " +
                                          nameSorted.First().ToString());

                  Console.WriteLine("Last in list " +
                                          nameSorted.Last().ToString());
            }
      }
}
```

Querying an array



Generic collections

- In our example programs so far we have already seen a *generic method PrintArray<T>*
  - This function outputs the string representation of the elements of an array
  - Its full declarations is :
    - *PrintArray<T>* takes any type which implements the *IEnumerable<T>* interface

```
public static void PrintArray<T>(IEnumerable<T> arr ,string message)
```

    - Thus an *IEnumerable* object of any type can be passed to the method

Generic collections

- The compiler infers the type *T* from the actual call to *PrintArray<T>*

```
public class LINQtoObjectArray
{
       static void Main(string[] args)
       {
              StudentInfo[] students ={……};

              // Order by last name and then first name
               var nameSorted =
              from s in students
              orderby s.LastName, s.FirstName
              select s;

              // PrintArray<StudentInfo> called here

              PrintArray(nameSorted, "Students sorted in last name, first name
                                        order");


              .
              .
       }
}
```

Generic collections

- Also the compiler determines whether the operations in the method body can be performed on any type the *T* represents

```
class MyClass
{

    public static void PrintArray(IEnumerable<int> arr, string message)
    {
            Console.Write("{0}",message);

            // Requires a ToString() override method for non built in types
            foreach (var element in arr)
                  Console.Write(" {0}", element);

            Console.WriteLine();
    }
}
```

Generic collections

- Collections store groups of objects
- We are all familiar with arrays
  - Arrays don't resize dynamically but do so when the *Resize()* method is called
- The collection class *List<T>* dynamically resizes when objects are inserted
  - It's known as a generic class because real classes are instantiated by providing actual types in place of *T*
    - *List<int>, List<string>, List<StudentInfo> etc*

Generic collections

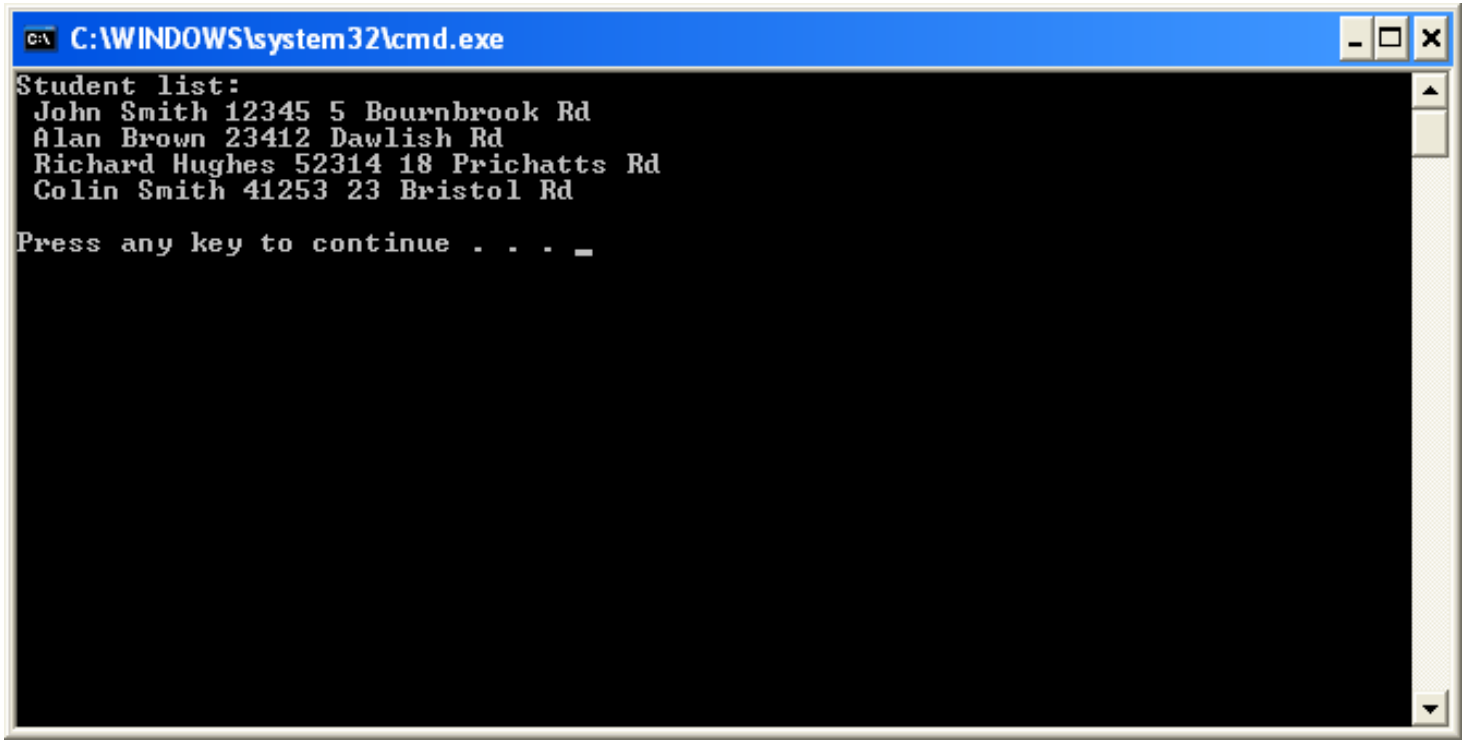
Method or property	Description
`Add`	Adds an element to the end of the List
`Capacity`	Property that gets or sets the number of elements a List can store
`Clear`	Removes all the elements from the List
`Contains`	Returns true if the List contains the specified element; otherwise, returns false
`Count`	Property that returns the number of elements stored in the List
`IndexOf`	Returns the index of the first occurrence of the specified value in the List
`Insert`	Inserts an element at the specified index
`Remove`	Removes the first occurrence of the specified value
`RemoveAt`	Removes the element at the specified index
`RemoveRange`	Removes a specified number of elements starting at a specified index
`Sort`	Sorts the List
`TrimExcess`	Sets the Capacity of the List to the number of elements the List currently contains (Count)

Generic collections

- We can create a list of *StudentInfo* objects and add items (to the end of the list) and insert items (anywhere in the list)
- We can display the list using exactly the same generic function as for displaying an array
- We could manipulate the list using the methods shown in the table

```
public class LINQtoList
{
      static void Main(string[] args)
      {

            StudentInfo[] students ={
                  new StudentInfo("Smith", "John", 12345, "5 Bournbrook Rd"),
                  new StudentInfo("Brown", "Alan", 23412, “Dawlish Rd"),
                  new StudentInfo("Smith","Colin", 41253, "23 Bristol Rd"),
                  new StudentInfo("Hughes", "Richard", 52314, "18 Prichatts Rd"),
                  new StudentInfo("Murphy", "Paul", 16352, "37 College Rd") };

            List<StudentInfo> studentList = new List<StudentInfo>();

            studentList.Add(students[0]);
            studentList.Add(students[1]);
            studentList.Add(students[2]);

            studentList.Insert(2, students[3]);

            PrintList(studentList, "Student list:");
      }

      public static void PrintList<T>(IEnumerable<T> arr, string message)
      {
            Console.WriteLine("{0}", message);

            foreach (T element in arr)
                  Console.WriteLine(" {0}", element);

            Console.WriteLine();
      }
}
```

Generic collections

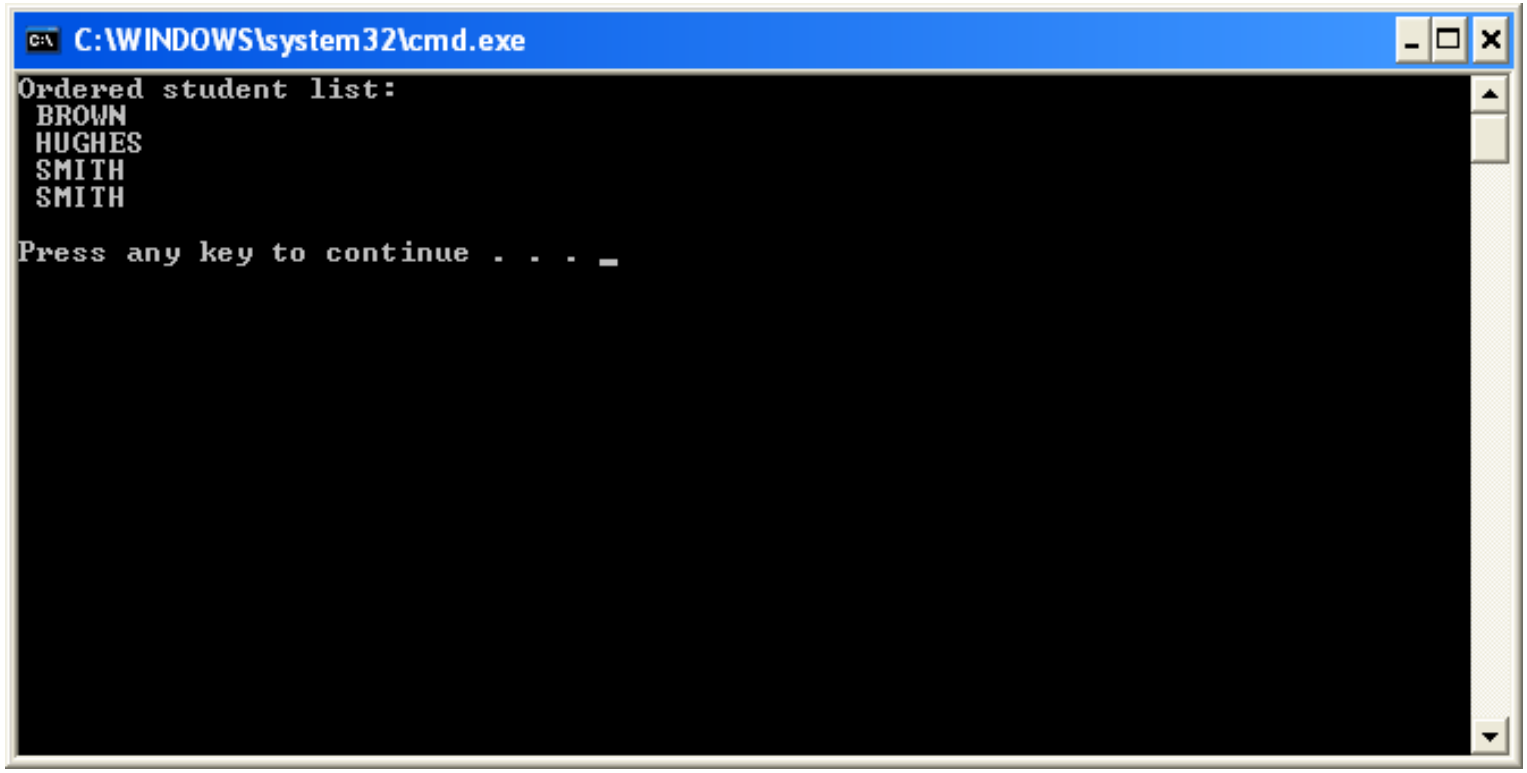


Querying a generic collection

- LINQ to Objects can query lists (and any other collection) in much the same way as querying an array
- For example, we could sort our list of students after first converting their surnames into upper case
  - The query makes use of the *let* clause which creates a new range variable
  - Enables a temporary result to be stored for later use in the query

```
public class LINQtoList
{
      static void Main(string[] args)
      {

            StudentInfo[] students ={
                  new StudentInfo("Smith", "John", 12345, "5 Bournbrook Rd"),
                  new StudentInfo("Brown", "Alan", 23412, “Dawlish Rd"),
                  new StudentInfo("Smith","Colin", 41253, "23 Bristol Rd"),
                  new StudentInfo("Hughes", "Richard", 52314, "18 Prichatts Rd"),
                  new StudentInfo("Murphy", "Paul", 16352, "37 College Rd") };

            List<StudentInfo> studentList = new List<StudentInfo>();

            studentList.Add(students[0]);
            studentList.Add(students[1]);
            studentList.Add(students[2]);

            studentList.Insert(2, students[3]);

             var orderedUpperCaseList =
                  from student in studentList
                  let upperCaseName = student.LastName.ToUpper()
                  orderby upperCaseName
                  select upperCaseName;

            PrintList(orderedUpperCaseList, "Ordered student list:");
      }

      public static void PrintList<T>(IEnumerable<T> arr, string message)
      {…}
}
```

Querying a generic collection



By

Dr. Mike Spann