9.1 Introduction - LINQ and List Collection

- A `List` is similar to an array but provides additional functionality, such as **dynamic resizing**.
- A language called SQL is the international standard used to perform **queries** (i.e., to request information that satisfies given criteria) and to manipulate data.
- C#'s **LINQ** (**Language-Integrated Query**) capabilities allow you to write **query expressions** that retrieve information from a *variety* of data sources, not just databases.
- **LINQ to Objects** can be used to **filter** arrays and `Lists`, selecting elements that satisfy a set of conditions

9.1 Introduction (Cont.)

- Figure 9.1 shows where and how we use LINQ throughout the book to retrieve information from many data sources.
- A **LINQ provider** is a set of classes that implement LINQ operations and enable programs to interact with *data sources* to perform tasks such as projecting, *sorting*, *grouping* and *filtering* elements.

9.1 Introduction (Cont.)

- There are two LINQ approaches
  - One uses a SQL-like syntax
  - The other uses method-call syntax.
  - This chapter shows the simpler SQL-like syntax.
  - Another chapter shows the method-call syntax, introducing the notions of delegates and lambdas—mechanisms that enable you to pass methods to other methods to help them perform their tasks.

9.2 Querying an Array of int Values Using LINQ

- Figure 9.2 shows how to use *LINQ to Objects* to query an array of integers, selecting elements that satisfy a set of conditions—a process called **filtering**
- Iteration statements that filter arrays focus on the steps required to get the results. This is called **imperative programming**.
- LINQ queries specify the conditions that selected elements must satisfy. This is known as **declarative programming**.
- The `System.Linq` namespace contains the LINQ to Objects provider.

```
1  // Fig. 9.2: LINQWithSimpleTypeArray.cs
2  // LINQ to Objects using an int array.
3  using System;
4  using System.Linq;
5
6  class LINQWithSimpleTypeArray
7  {
8     static void Main()
9     {
10       // create an integer array
11       var values = new[] {2, 9, 5, 0, 3, 7, 1, 4, 8, 5};
12
13       // display original values
14       Console.Write("Original array:");
15       foreach (var element in values)
16       {
17          Console.Write($" {element}");
18       }
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 1 of 6.)

```
19
20          // LINQ query that obtains values greater than 4 from the array
21          var filtered =
22             from value in values // data source is values
23             where value > 4
24             select value;
25
26          // display filtered results
27          Console.Write("\nArray values greater than 4:");
28          foreach (var element in filtered)
29          {
30             Console.Write($" {element}");
31          }
32
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 2 of 6.)

```
33          // use orderby clause to sort original values in ascending order
34          var sorted =
35             from value in values // data source is values
36             orderby value
37             select value;
38
39          // display sorted results
40          Console.Write("\nOriginal array, sorted:");
41          foreach (var element in sorted)
42          {
43             Console.Write($" {element}");
44          }
45
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 3 of 6.)

```
46          // sort the filtered results into descending order
47          var sortFilteredResults =
48             from value in filtered  // data source is LINQ query filtered
49             orderby value descending
50             select value;
51
52          // display the sorted results
53          Console.Write(
54             "\nValues greater than 4, descending order (two queries):");
55          foreach (var element in sortFilteredResults)
56          {
57             Console.Write($" {element}");
58          }
59
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 4 of 6.)

```
60          // filter original array and sort results in descending order
61          var sortAndFilter =
62             from value in values     // data source is values
63             where value > 4
64             orderby value descending
65             select value;
66
67          // display the filtered and sorted results
68          Console.Write(
69             "\nValues greater than 4, descending order (one query):");
70          foreach (var element in sortAndFilter)
71          {
72             Console.Write($" {element}");
73          }
74
75          Console.WriteLine();
76       }
77    }
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 5 of 6.)

```
Original array: 2 9 5 0 3 7 1 4 8 5
Array values greater than 4: 9 5 7 8 5
Original array, sorted: 0 1 2 3 4 5 5 7 8 9
Values greater than 4, descending order (two queries): 9 8 7 5 5
Values greater than 4, descending order (one query): 9 8 7 5 5
```

Fig. 9.2 | LINQ to Objects using an `int` array. (Part 6 of 6.)

9.2.1 The from Clause

- A LINQ query begins with a **`from` clause**, which specifies a **range variable** (`value`) and the data source to query (`values`).
  - The range variable represents each item in the data source, much like the control variable in a `foreach` statement.

9.2.2 The where Clause

- If the condition in the `where clause` evaluates to `true`, the element is selected.
- A **predicate** is an expression that takes an element of a collection and returns `true` or `false` by testing a condition on that element.

9.2.3 The select Clause

- The **select clause** determines what value appears in the results.

9.2.5 The orderby Clause

- The **orderby clause** sorts the query results in ascending order.
- The **descending** modifier in the `orderby` clause sorts the results in descending order.
- Any value that can be compared with other values of the same type may be used with the `orderby` clause.

9.2.6 Interface IEnumerable<T>

- The `IEnumerable<T>` interface describes the functionality of any object that can be iterated over and thus offers members to access each element.
- Arrays and collections already implement the `IEnumerable<T>` interface.
- A LINQ query returns an object that implements the `IEnumerable<T>` interface.
- With LINQ, the code that selects elements and the code that displays them are kept separate, making the code easier to understand and maintain.

9.3 Querying an Array of Employee Objects Using LINQ

- LINQ is not limited to querying arrays of simple types such as `integers`.
- It cannot be used when a query does not have a defined meaning—for example, you cannot use orderby on objects values that are not *comparable*.
- Comparable types in .NET are those that implement the `IComparable<T>`.
- All built-in types, such as `string`, `int` and double implement `IComparable<T>`.
- Figure 9.3 presents the Employee class. Figure 9.4 uses LINQ to query an array of Employee objects.

```
1  // Fig. 9.3: Employee.cs
2  // Employee class with FirstName, LastName and MonthlySalary properties.
3  class Employee
4  {
5     public string FirstName { get; } // read-only auto-implemented property
6     public string LastName { get; } // read-only auto-implemented property
7     private decimal monthlySalary; // monthly salary of employee
8
9     // constructor initializes first name, last name and monthly salary
10    public Employee(string firstName, string lastName,
11       decimal monthlySalary)
12    {
13       FirstName = firstName;
14       LastName = lastName;
15       MonthlySalary = monthlySalary;
16    }
17
```

Fig. 9.3 | `Employee` class with `FirstName`, `LastName` and `MonthlySalary` properties. (Part 1 of 2.)

```
18      // property that gets and sets the employee's monthly salary
19      public decimal MonthlySalary
20      {
21         get
22         {
23            return monthlySalary;
24         }
25         set
26         {
27            if (value >= 0M) // validate that salary is nonnegative
28            {
29               monthlySalary = value;
30            }
31         }
32      }
33
34      // return a string containing the employee's information
35      public override string ToString() =>
36         $"{FirstName,-10} {LastName,-10} {MonthlySalary,10:C}";
37   }
```

Fig. 9.3 | Employee class with FirstName, LastName and MonthlySalary properties. (Part 2 of 2.)

```
1  // Fig. 9.4: LINQWithArrayOfObjects.cs
2  // LINQ to Objects querying an array of Employee objects.
3  using System;
4  using System.Linq;
5
6  class LINQWithArrayOfObjects
7  {
8     static void Main()
9     {
10       // initialize array of employees
11       var employees = new[] {
12          new Employee("Jason", "Red", 5000M),
13          new Employee("Ashley", "Green", 7600M),
14          new Employee("Matthew", "Indigo", 3587.5M),
15          new Employee("James", "Indigo", 4700.77M),
16          new Employee("Luke", "Indigo", 6200M),
17          new Employee("Jason", "Blue", 3200M),
18          new Employee("Wendy", "Brown", 4236.4M)};
```

Fig. 9.4 | LINQ to Objects querying an array of `Employee` objects. (Part 1 of 7.)

```
19
20          // display all employees
21          Console.WriteLine("Original array:");
22          foreach (var element in employees)
23          {
24             Console.WriteLine(element);
25          }
26
27          // filter a range of salaries using && in a LINQ query
28          var between4K6K =
29             from e in employees
30             where (e.MonthlySalary >= 4000M) && (e.MonthlySalary <= 6000M)
31             select e;
32
33          // display employees making between 4000 and 6000 per month
34          Console.WriteLine("\nEmployees earning in the range" +
35             $"{4000:C}-{6000:C} per month:");
36          foreach (var element in between4K6K)
37          {
38             Console.WriteLine(element);
39          }
```

Fig. 9.4 | LINQ to Objects querying an array of `Employee` objects. (Part 2 of 7.)

```
40
41          // order the employees by last name, then first name with LINQ
42          var nameSorted =
43             from e in employees
44             orderby e.LastName, e.FirstName
45             select e;
46
47          // header
48          Console.WriteLine("\nFirst employee when sorted by name:");
49
50          // attempt to display the first result of the above LINQ query
51          if (nameSorted.Any())
52          {
53             Console.WriteLine(nameSorted.First());
54          }
55          else
56          {
57             Console.WriteLine("not found");
58          }
```

Fig. 9.4 | LINQ to Objects querying an array of `Employee` objects. (Part 3 of 7.)

```
59
60          // use LINQ to select employee last names
61          var lastNames =
62             from e in employees
63             select e.LastName;
64
65          // use method Distinct to select unique last names
66          Console.WriteLine("\nUnique employee last names:");
67          foreach (var element in lastNames.Distinct())
68          {
69             Console.WriteLine(element);
70          }
71
```

Fig. 9.4 | LINQ to Objects querying an array of `Employee` objects. (Part 4 of 7.)

```
72          // use LINQ to select first and last names
73          var names =
74             from e in employees
75             select new {e.FirstName, e.LastName};
76
77          // display full names
78          Console.WriteLine("\nNames only:");
79          foreach (var element in names)
80          {
81             Console.WriteLine(element);
82          }
83
84          Console.WriteLine();
85       }
86    }
```

Fig. 9.4 | LINQ to Objects querying an array of `Employee` objects. (Part 5 of 7.)

```
Original array:
Jason       Red          $5,000.00
Ashley      Green        $7,600.00
Matthew     Indigo       $3,587.50
James       Indigo       $4,700.77
Luke        Indigo       $6,200.00
Jason       Blue         $3,200.00
Wendy       Brown        $4,236.40

Employees earning in the range $4,000.00-$6,000.00 per month:
Jason       Red          $5,000.00
James       Indigo       $4,700.77
Wendy       Brown        $4,236.40

First employee when sorted by name:
Jason       Blue         $3,200.00
```

Fig. 9.4 | LINQ to Objects querying an array of `Employee` objects. (Part 6 of 7.)

```
Unique employee last names:
Red
Green
Indigo
Blue
Brown

Names only:
{ FirstName = Jason, LastName = Red }
{ FirstName = Ashley, LastName = Green }
{ FirstName = Matthew, LastName = Indigo }
{ FirstName = James, LastName = Indigo }
{ FirstName = Luke, LastName = Indigo }
{ FirstName = Jason, LastName = Blue }
{ FirstName = Wendy, LastName = Brown }
```

Fig. 9.4 | LINQ to Objects querying an array of `Employee` objects. (Part 7 of 7.)

9.3.1 Accessing the Properties of a LINQ Query's Range Variable

- A `where` clause can access the properties of the range variable.
- The conditional AND (&&) operator can be used to combine conditions.

9.3.2 Sorting a LINQ Query's Results by Multiple Properties

- An `orderby` clause can sort the results according to multiple properties, specified in a comma-separated list.

9.3.3 Any, First and Count Extension Methods

- The query result's `Any` method returns true if there is at least one element, and false if there are no elements.
- The query result's `First` method returns the first element in the result.
- The `Count` method of the query result returns the number of elements in the results.

9.3.4 Selecting a Property of an Object

- The `select` clause can be used to select a member of the range variable rather than the range variable itself.
- The `Distinct` method removes duplicate elements, causing all elements in the result to be unique.

9.3.5 Creating New Types in the select Clause of a LINQ Query

- The `select` clause can create a new object of **anonymous type** (a type with no name), which the compiler generates for you based on the properties listed in the curly braces ({}).
  - `new {e.FirstName, e.LastName}`
- By default, the name of the property being selected is used as the property's name in the result.
- You can specify a different name for the property inside the anonymous type definition

9.3.5 Creating New Types in the select Clause of a LINQ Query

- Implicitly typed local variables allow you to use anonymous types because you do not have to explicitly state the type when declaring such variables.
- When the compiler creates an anonymous type, it automatically generates a `ToString` method that returns a `string` representation of the object.

9.3.5 Creating New Types in the select Clause of a LINQ Query

- Creating an anonymous type in a LINQ quere is an example of a **projection**—it performs a transformation on the data.
- The transformation creates new objects containing only the specified properties
- Such rransformations can also manipulate the data.
  - For example, you could give all employees a 10% raise by multiplying their `MonthlySalary` properties by 1.1.

9.4 Introduction to Collections

- The .NET Framework Class Library provides several classes, called *collections*, used to store groups of related objects.
- These classes provide efficient methods that organize, store and retrieve your data *without* requiring knowledge of how the data is being *stored*.

9.4.1 List<T> Collection

- The generic collection class `List<T>` (from namespace `System.Collections.Generic`) does not need to be reallocated to change its size.
- `List<T>` is called a **generic class** because it can be used with any type of object.
- `T` is a placeholder for the type of the objects stored in the list.
- Figure 9.5 shows some common methods and properties of class `List<T>`.

Method or property	Description
`Add`	Adds an element to the end of the `List`.
`AddRange`	Adds the elements of its collection argument to the end of the `List`.
`Capacity`	Property that *gets* or *sets* the number of elements a `List` can store without resizing.
`Clear`	Removes all the elements from the `List`.
`Contains`	Returns `true` if the `List` contains the specified element and `false` otherwise.
`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`.

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 1 of 2.)

Method or property	Description
`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`).

Fig. 9.5 | Some methods and properties of class `List<T>`. (Part 2 of 2.)

9.4.2 Dynamically Resizing a List<T> Collection

- Figure 9.6 demonstrates dynamically resizing a `List` object.
- The `Count` property returns the number of elements currently in the List.
- The `Capacity` property indicates how many items the `List` can hold without having to grow.
- When the `List` is created, both are initially 0—though the `Capacity` is implementation dependent.

```
1  // Fig. 9.6: ListCollection.cs
2  // Generic List<T> collection demonstration.
3  using System;
4  using System.Collections.Generic;
5
6  class ListCollection
7  {
8     static void Main()
9     {
10       // create a new List of strings
11       var items = new List<string>();
12
13       // display List's Count and Capacity before adding elements
14       Console.WriteLine("Before adding to items: " +
15          $"Count = {items.Count}; Capacity = {items.Capacity}");
16
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 1 of 7.)

```
17          items.Add("red"); // append an item to the List
18          items.Insert(0, "yellow"); // insert the value at index 0
19
20          // display List's Count and Capacity after adding two elements
21          Console.WriteLine("After adding two elements to items: " +
22             $"Count = {items.Count}; Capacity = {items.Capacity}");
23
24          // display the colors in the list
25          Console.Write(
26             "\nDisplay list contents with counter-controlled loop:");
27          for (var i = 0; i < items.Count; i++)
28          {
29             Console.Write($" {items[i]}");
30          }
31
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 2 of 7.)

```
32          // display colors using foreach
33          Console.Write("\nDisplay list contents with foreach statement:");
34          foreach (var item in items)
35          {
36             Console.Write($" {item}");
37          }
38
39          items.Add("green"); // add "green" to the end of the List
40          items.Add("yellow"); // add "yellow" to the end of the List
41
42          // display List's Count and Capacity after adding two more elements
43          Console.WriteLine("\n\nAfter adding two more elements to items: " +
44             $"Count = {items.Count}; Capacity = {items.Capacity}");
45
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 3 of 7.)

```
46          // display the List
47          Console.Write("\nList with two new elements:");
48          foreach (var item in items)
49          {
50             Console.Write($" {item}");
51          }
52
53          items.Remove("yellow"); // remove the first "yellow"
54
55          // display the List
56          Console.Write("\n\nRemove first instance of yellow:");
57          foreach (var item in items)
58          {
59             Console.Write($" {item}");
60          }
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 4 of 7.)

```
61
62          items.RemoveAt(1); // remove item at index 1
63
64          // display the List
65          Console.Write("\nRemove second list element (green):");
66          foreach (var item in items)
67          {
68             Console.Write($" {item}");
69          }
70
71          // display List's Count and Capacity after removing two elements
72          Console.WriteLine("\nAfter removing two elements from items: " +
73             $"Count = {items.Count}; Capacity = {items.Capacity}");
74
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 5 of 7.)

```
75          // check if a value is in the List
76          Console.WriteLine("\n\"red\" is " +
77             $"{(items.Contains("red") ? string.Empty : "not ")}in the list");
78
79          items.Add("orange"); // add "orange" to the end of the List
80          items.Add("violet"); // add "violet" to the end of the List
81          items.Add("blue"); // add "blue" to the end of the List
82
83          // display List's Count and Capacity after adding three elements
84          Console.WriteLine("\nAfter adding three more elements to items: " +
85             $"Count = {items.Count}; Capacity = {items.Capacity}");
86
87          // display the List
88          Console.Write("List with three new elements:");
89          foreach (var item in items)
90          {
91             Console.Write($" {item}");
92          }
93          Console.WriteLine();
94       }
95    }
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 6 of 7.)

```
Before adding to items: Count = 0; Capacity = 0
After adding two elements to items: Count = 2; Capacity = 4

Display list contents with counter-controlled loop: yellow red
Display list contents with foreach statement: yellow red

After adding two more elements to items: Count = 4; Capacity = 4
List with two new elements: yellow red green yellow

Remove first instance of yellow: red green yellow
Remove second list element (green): red yellow
After removing two elements from items: Count = 2; Capacity = 4

"red" is in the list

After adding three more elements to items: Count = 5; Capacity = 8
List with three new elements: red yellow orange violet blue
```

Fig. 9.6 | Generic `List<T>` collection demonstration. (Part 7 of 7.)

9.4 Introduction to Collections (Cont.)

- The `Add` method *appends* its argument to the end of the `List`.
- The `Insert` method inserts a new element at the specified position.
  - The first argument is an index—as with arrays, collection indices start at zero.
  - The second argument is the value that's to be inserted at the specified index.
  - The indices of elements at the specified index and above increase by one.

9.4 Introduction to Collections (Cont.)

- `Lists` can be indexed like arrays by placing the index in square brackets after the `List` variable's name.
- The `Remove` method is used to remove the *first* instance of an element with a specific value.
  - If no such element is in the List, `Remove` does nothing.
- `RemoveAt` removes the element at the specified index; the indices of all elements above that index decrease by one.

9.4 Introduction to Collections (Cont.)

- The `Contains` method returns `true` if the element is found in the `List`, and `false` otherwise.
- `Contains` compares its argument to each element of the `List` in order, so using `Contains` on a large `List` is inefficient.
- When the `List` grows, it must create a larger internal array and copy each element to the new array.
- A `List` grows only when an element is added and there is no space for the new element.

Performance Tip 9.1

Doubling a `List`*'s* `Capacity` *is an efficient way for a* `List` *to grow quickly to be "about the right size." This operation is much more efficient than growing a* `List` *by only as much space as it takes to hold the element(s) being added. A disadvantage is that the* `List` *might occupy more space than it requires. This is a classic example of the space/time trade-off.*

Performance Tip 9.2

It can be wasteful to double a `List`*'s size when more space is needed. For example, a full* `List` *of 1,000,000 elements resizes to accommodate 2,000,000 elements when one new element is added. This leaves 999,999 unused elements. You can use* `TrimExcess` *(as in* `yourListObject.TrimExcess()`*) to reduce a* `List`*'s* `Capacity` *to its current* `Count`*. You also can set the* `Capacity` *directly to control space usage better—for example, if you know a* `List` *will never grow beyond 100 elements, you can preallocate that space by assigning 100 to the* `List`*'s* `Capacity` *or using the* `List` *constructor that receives an initial capacity.*

9.5 Querying a Generic Collection Using LINQ

- You can use LINQ to Objects to query Lists just as arrays.
- In Fig. 9.7, a List of strings is converted to uppercase and searched for those that begin with "R".

```
1  // Fig. 9.7: LINQWithListCollection.cs
2  // LINQ to Objects using a List<string>.
3  using System;
4  using System.Linq;
5  using System.Collections.Generic;
6
7  class LINQWithListCollection
8  {
9     static void Main()
10    {
11       // populate a List of strings
12       var items = new List<string>();
13       items.Add("aQua"); // add "aQua" to the end of the List
14       items.Add("RusT"); // add "RusT" to the end of the List
15       items.Add("yElLow"); // add "yElLow" to the end of the List
16       items.Add("rEd"); // add "rEd" to the end of the List
17
18       // display initial List
19       Console.Write("items contains:");
20       foreach (var item in items)
21       {
22          Console.Write($" {item}");
23       }
```

Fig. 9.7 | LINQ to Objects using a `List<string>`. (Part 1 of 4.)

```
24
25          Console.WriteLine(); // output end of line
26
27          // convert to uppercase, select those starting with "R" and sort
28          var startsWithR =
29             from item in items
30             let uppercaseString = item.ToUpper()
31             where uppercaseString.StartsWith("R")
32             orderby uppercaseString
33             select uppercaseString;
34
35          // display query results
36          Console.Write("results of query startsWithR:");
37          foreach (var item in startsWithR)
38          {
39             Console.Write($" {item}");
40          }
41
42          Console.WriteLine(); // output end of line
```

Fig. 9.7 | LINQ to Objects using a `List<string>`. (Part 2 of 4.)

```
43
44          items.Add("rUbY"); // add "rUbY" to the end of the List
45          items.Add("SaFfRon"); // add "SaFfRon" to the end of the List
46
47          // display initial List
48          Console.Write("items contains:");
49          foreach (var item in items)
50          {
51             Console.Write($" {item}");
52          }
53
54          Console.WriteLine(); // output end of line
55
56          // display updated query results
57          Console.Write("results of query startsWithR:");
58          foreach (var item in startsWithR)
59          {
60             Console.Write($" {item}");
61          }
62
63          Console.WriteLine(); // output end of line
64       }
65    }
```

Fig. 9.7 | LINQ to Objects using a `List<string>`. (Part 3 of 4.)

```
items contains: aQua RusT yElLow rEd
results of query startsWithR: RED RUST
items contains: aQua RusT yElLow rEd rUbY SaFfRon
results of query startsWithR: RED RUBY RUST
```

Fig. 9.7 | LINQ to Objects using a `List<string>`. (Part 4 of 4.)

9.5.1 The let Clause

- LINQ's `let clause` can be used to create a new range variable to store a temporary result for use later in the LINQ query.
- The `string` method `ToUpper` to converts a string to uppercase.
- The `string` method `StartsWith` performs a case sensitive comparison to determine whether a `string` starts with the `string` received as an argument.

9.5.2 Deferred Execution

- LINQ uses **deferred execution**—the query executes *only* when you access the results, *not* when you define the query.

Performance Tip 9.3

Deferred execution can improve performance when a query's results are not immediately needed.

9.5.3 Extension Methods ToArray and ToList

- LINQ extension methods `ToArray` and `ToList` immediately execute the query on which they are called.
  - These methods execute the query only once, improving efficiency.

Performance Tip 9.4

Methods `ToArray` and `ToList` also can improve efficiency if you'll be iterating over the same results multiple times, as you execute the query only once.

9.5.4 Collection Initializers

- **Collection initializers** provide a convenient syntax (similar to *array initializers*) for initializing a collection. For example, lines 12–16 of Fig. 9.7 could be replaced with:
  - `var items = new List<string> {"aQua", "RusT", "yElLow", "rEd"};`
  - Explicitly creates the `List<string>` with `new`, so the compiler knows that the initializer list contains elements for a `List<string>`
- The following declaration would generate a compilation error, because the compiler cannot determine whether you wish to create an array or a collection
  - `var items = {"aQua", "RusT", "yElLow", "rEd"};`