Chapter 22. Lambda Expressions and LINQ

In This Chapter

In this chapter we will become acquainted with some of the advanced capabilities of the C# language. To be more specific, we will pay attention on how to make **queries to collections**, using **lambda expressions** and **LINQ**, and how to add functionality to already created classes, using **extension methods**. We will get to know the **anonymous types**, describe their usage briefly and discuss lambda expressions and show in practice how most of the built-in **lambda functions** work. Afterwards, we will pay more attention to the LINQ syntax – we will learn what it is, how it works and what queries we can build with it. In the end, we will get to know the meaning of the **keywords in LINQ**, and demonstrate their capabilities with lots of examples.

Extension Methods

In practice, programmers often have to **add new functionality** to already existing code. If the code is available, we can simply add the required functionality and recompile. When a given assembly (**`.exe`** or **`.dll`** file) has already been compiled, and the source code is not available, a common way to extend the functionality of the types is trough inheritance. This approach can be quite difficult to apply, due to the fact that we will have to change the instances of the base class with the instances of the derived one to be able to use our new functionality. Unfortunately, that is the least of our problems. If the type we want to inherit is marked with the keyword **`sealed`**, inheritance is not possible.

Extension methods solve that very same problem – they present to us the opportunity to **add new functionality to already existing type** (class or interface), without having to change its original code or use inheritance, i.e. also works fine with types that cannot be inherited. Notice that trough extension methods we can add "implemented methods" even to interfaces.

The extension methods are defined as **`static`** in ordinary static classes. The type of their first argument is the class (or the interface) they extend. In front of it, we should place **the keyword `this`**. That is what makes them different from other static methods, and indicates the compiler that this is an extension method. The parameter with the keyword **`this`** in front of it can be used in the method body to create its functionality. Practically, it is the object that is used by the extension method.

Extension methods can be applied directly to objects of the class/interface they extend. They can also be invoked statically through the static class they are defined in, but it is not a good practice.

To refer to a specific extension method, we should add "using" and the corresponding namespace, where the static class, describing this method, is defined. Otherwise the compiler has no way of knowing about their existence.

Extension Methods – Examples

Let's take for example the **definition of an extension method** that counts the number of words in a given string. Have in mind, that the type **string** is **sealed**, so it cannot be inherited.

```
public static class StringExtensions
{
   public static int WordCount(this string str)
   {
      return str.Split(new char[] { ' ', '.', '?', '!' },
         StringSplitOptions.RemoveEmptyEntries).Length;
   }
}
```

The method **WordCount(…)** extends the class **String**. This is indicated by the keyword **this** before the type and the name of the first argument of the method (in our case **str**). The method itself is **static** and it is defined in the static class **StringExtensions**. The usage of the extension method is done the same way as all the other methods of the class **String**. Do not forget to add the corresponding namespace, where the static class, describing the extension methods, is defined. Example of **using an extension method**:

```
static void Main()
{
   string helloString = "Hello, Extension Methods!";
   int wordCount = helloString.WordCount();
   Console.WriteLine(wordCount);
}
```

The method is invoked on the object **helloString**, which is of type **string**. It also takes the object as an argument and works with it (in our case refers to its **Split(…)** method and returns the number of elements of the array, produced by the **Split(…)** method).

Extension Methods for Interfaces

Extension methods can not only be used on classes, but on interfaces as well. Our next example takes an instance of a class, that implements the interface list of integers (**IList<int>**), and increases their value by a certain number. The method **IncreaseWith(…)** can access only those elements that are included in the interface **IList** (e.g. the property **Count**).

```
public static class IListExtensions
{
   public static void IncreaseWith(
      this IList<int> list,   int amount)
   {
      for (int i = 0; i < list.Count; i++)
      {
         list[i] += amount;
      }
   }
}
```

The extension methods also give us the opportunity to work on generic types. Let's take for example a method that loops trough a collection, using **foreach**, implementing **IEnumerable** from generic type **T**. Its purpose is to convert to a meaningful string a sequence of elements (e.g. a list of integers):

```
public static class IEnumerableExtensions
{
   public static string ToString<T>(
      this IEnumerable<T> enumeration)
   {
      StringBuilder result = new StringBuilder();
      result.Append("[");

      foreach (var item in enumeration)
      {
         result.Append(item.ToString());
         result.Append(", ");
      }

      if (result.Length > 1)
         result.Remove(result.Length - 2, 2);
      result.Append("]");
      return result.ToString();
   }
}
```

Example of how to use the two extension methods declared above:

```
static void Main()
{
   List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
   Console.WriteLine(numbers.ToString<int>());
   numbers.IncreaseWith(5);
   Console.WriteLine(numbers.ToString<int>());
}
```

The **output** of the execution of the program will be the following:

```
[1, 2, 3, 4, 5]
[6, 7, 8, 9, 10]
```

Anonymous Types

In object-oriented languages (such as C#), it is common to define small classes that will be used only once. Typical example is the class **Point** that has only two fields – the coordinates of a point. Creating a simple class with the idea of using it just once is inconvenient and time consuming for the programmer, especially when the standard operations for each class: **ToString()**, **Equals()** and **GetHashCode()** have to be predefined.

In C# there is a built-in way to create **single-use types**, called **anonymous types**. Objects of such type are created almost the same way as other objects in C#. The thing with them is that we don't need to define data type for the variable in advance. The **keyword var** indicates to the compiler that the type of the variable will be automatically detected by the expression, after the equals sign. We actually don't have a choice here, since we can't tell the specific type of the variable, because it is defined as one of an **anonymous type**. After that, we specify name for the object, followed by the "=" operator and the keyword **new**. In curly braces we enumerate the names and the values of the properties of the anonymous type.

Anonymous Types – Example

Here is an **example of creating an anonymous type** that describes a car:

```
var myCar = new { Color = "Red", Brand = "BMW", Speed = 180 };
```

During compilation, the compiler will create a class with a **unique name** (something like **<>f__AnonymousType0**) and will generate properties for it (with getter and setter). In the example above, the compiler will guess by its own, that the properties **Color** and **Brand** are of type **string** and **Speed** will be set as **int**. Right after the initialization, the object of the anonymous type can be used as one of an ordinary type with its three properties:

```
Console.WriteLine("My car is a {0} {1}.",
   myCar.Color, myCar.Brand);
Console.WriteLine("It runs {0} km/h.", myCar.Speed);
```

The output of the code above will be as follows:

```
My car is a Red BMW.
It runs 180 km/h.
```

More about Anonymous Types

As any other type in .NET, the anonymous ones inherit the class **System.Object**. During compilation, the compiler will automatically redefine the methods **ToString()**, **Equals()** and **GetHashCode()** for us.

```
Console.WriteLine("ToString: {0}", myCar.ToString());
Console.WriteLine("Hash code: {0}",
   myCar.GetHashCode().ToString());
Console.WriteLine("Equals? {0}", myCar.Equals(
   new { Color = "Red", Brand = "BMW", Speed = 180 }
));
Console.WriteLine("Type name: {0}", myCar.GetType().ToString());
```

The output of the code above will be the following:

```
ToString: { Color = Red, Brand = BMW, Speed = 180 }
Hash code: 1572002086
Equals? True
Type name:
<>f__AnonymousType0`3[System.String,System.String,System.Int32]
```

As we can see from the result, the method **ToString() is redefined**, so that it can list the properties of the anonymous type in the order of their definition in the initialization of the object (in our case **myCar**). The method **GetHashCode()** is wrote in such a way, that it uses all fields and on their basis it calculates a hash function with a small number of collisions. The redefined by the compiler method **Equals(…)** compares the objects field by field. As we can notice from the example, we have created a new object that has exactly the same properties as **myCar**, and returns a result stating that the newly created object and the old one have equal values.

Arrays of Anonymous Types

The anonymous types, like ordinary ones, can be used as **elements of arrays**. We can initialize them with the keyword **new**, followed by square brackets. The values of the elements of the array are listed the same way, as

the values assigned to the anonymous types. The values in the array should be homogeneous, i.e. it is not possible to have different anonymous types in the same array. An example of defining an array of anonymous types with two properties (**X** and **Y**):

```
var arr = new[] {
   new { X = 3, Y = 5 },
   new { X = 1, Y = 2 },
   new { X = 0, Y = 7 }
};
foreach (var item in arr)
{
   Console.WriteLine(item.ToString());
}
```

The result of the execution of the code above will be the following:

```
{ X = 3, Y = 5 }
{ X = 1, Y = 2 }
{ X = 0, Y = 7 }
```

Lambda Expressions

Lambda expressions are anonymous functions that contain expressions or sequence of operators. All lambda expressions use the lambda operator **=>**, which can be read as "**goes to**". The idea of the lambda expressions in C# is borrowed from the functional programming languages (e.g. **Haskell**, **Lisp**, **Scheme**, **F#** and others). The left side of the lambda operator specifies the **input parameters** and the right side holds an **expression** or a code block that works with the entry parameters and conceivably returns some result.

Usually lambda expressions are used as **predicates** or instead of **delegates** (a type that references a method instance), which can be applied on collections, processing their elements and/or returning a certain result.

Lambda Expressions – Examples

As an example, let's take the **extension method `FindAll(…)`**, which can be used to filter the necessary elements. It works on a certain collection by applying a given **predicate** on it that checks if an element matches a certain requirement. In order to use it we have to add a reference to the assembly **`System.Core.dll`** (if it is not already added) and include the namespace **`System.Linq`**, because the extension methods for the collections are there.

For example, if we want to take only the even numbers from a collection of integers, we can use the method **`FindAll(…)`** on that collection, passing a lambda method to it that checks if a certain number is even:

```
List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);

foreach (var num in evenNumbers)
{
   Console.Write("{0} ", num);
}

Console.WriteLine();
```

The result is:

```
2 4 6
```

The example above loops through the whole collection of numbers and for each element (named **x**) a check, if the number is multiple of 2, is made (through the Boolean expression **(x % 2) == 0**).

Let's now focus on an example in which trough an **extension method** and a **lambda expression** we will create a collection, containing data from a certain class. In the example, from the class **Dog** (with properties **Name** and **Age**), we want to get a list that contains all dogs' names. We can do that with the **extension method Select(…)** (defined in the namespace **System.Linq**) by assigning to it to turn each dog (**x**) into dog's name (**x.Name**) and writing that result in the variable **names**. With the keyword **var**, we tell the compiler to define the type of the variable according to the result that we assign on the right side of the equals sign.

```
class Dog
{
   public string Name { get; set; }
   public int Age { get; set; }
}

static void Main()
{
   List<Dog> dogs = new List<Dog>() {
      new Dog { Name = "Rex", Age = 4 },
      new Dog { Name = "Sean", Age = 0 },
      new Dog { Name = "Stacy", Age = 3 }
   };
   var names = dogs.Select(x => x.Name);
   foreach (var name in names)
   {
      Console.WriteLine(name);
   }
```

```
}
```

The result is:

```
Rex
Sean
Stacy
```

Using Lambda Expressions with Anonymous Types

We can create **collections of anonymous types** from a collection with some elements by **using lambda expressions**. Let's take the collection **dogs**, containing elements of type **Dog**, and create new collection consisting of elements of an anonymous type, having two properties – age and the initial letter of the dog's name:

```
var newDogsList = dogs.Select(
   x => new { Age = x.Age, FirstLetter = x.Name[0] });
foreach (var item in newDogsList)
{
   Console.WriteLine(item);
}
```

The result is:

```
{ Age = 4, FirstLetter = R }
{ Age = 0, FirstLetter = S }
{ Age = 3, FirstLetter = S }
```

As it is obvious from the example above, the newly created collection **newDogsList** has elements of an anonymous type, taking the properties **Age** and **FirstLetter** as parameters. The first line of the example can be read as follows: "Create a variable of undefined (at this point) type, name it **newDogsList** and create a new element of an anonymous type for each element **x** of the **dogs** collection with two properties: **Age** that is equal to the property **Age** of the element **x**, and the property **FirstLetter** that is equal to the first character of the string **x.Name**".

Sorting with Lambda Expressions

If we want to sort the elements in a certain collection, we can use the extension methods **OrderBy(…)** and **OrderByDescending(…)**, by defining the way of sorting in a lambda function. An example on our collection **dogs**:

```
var sortedDogs = dogs.OrderByDescending(x => x.Age);
```

```
foreach (var dog in sortedDogs)
{
   Console.WriteLine(string.Format(
      "Dog {0} is {1} years old.", dog.Name, dog.Age));
}
```

The result is:

```
Dog Rex is 4 years old.
Dog Stacy is 3 years old.
Dog Sean is 0 years old.
```

Statements in Lambda Expressions

Lambda functions can also have a **body**. So far we have used lambda functions with only one statement. Now we will pay more attention to lambda functions that have a body. Let's return to the example with the even numbers. Suppose we want to print to the console the values of all numbers, to which our lambda function is applied to and to return the result if they are even or not. We can do it the following way:

```
List<int> list = new List<int>() { 20, 1, 4, 8, 9, 44 };
// Process each argument with code statements
var evenNumbers = list.FindAll((i) =>
{
   Console.WriteLine("Value of i is: {0}", i);
   return (i % 2) == 0;
});
```

The result from the above code is:

```
Value of i is: 20
Value of i is: 1
Value of i is: 4
Value of i is: 8
Value of i is: 9
Value of i is: 44
```

Lambda Expressions as Delegates

Lambda functions can be written in delegates. **Delegates** are such a type of variables that contains functions (methods). Some standard delegate types in .NET are: **Action**, **Action<in T>**, **Action<in T1, in T2>**, and so on and **Func<out TResult>, Func<in T, out TResult>, Func<in T1, in T2, out TResult>** and so on. The types **Func** and **Action** are generic and

contain the types of the return value, and the types of the parameters of the functions. The variables of such types are references to functions. Below is an example for using and assigning values to these types:

```
Func<bool> boolFunc = () => true;
Func<int, bool> intFunc = (x) => x < 10;
if (boolFunc() && intFunc(5))
{
  Console.WriteLine("5 < 10");
}
```

The result is:

```
5 < 10
```

In the example above we define **two delegates**. The first one – **boolFunc** is a function that has no input parameters and returns a Boolean result. We have given an anonymous lambda function that does nothing and always returns **true** as a value to that function. The second delegate **intFunc** takes as an argument an **int** variable and returns a Boolean value – true when **x** is less than ten, and false otherwise. At the end, in the **if** statement, we call these two delegates as we give to the second one value of 5 as an argument, and the result from their invocation is **true**, as we can see.

LINQ Queries

LINQ (Language-Integrated Query) is a set of extensions of the .NET Framework, that includes language integrated queries and operations on the elements of a certain **data source** (most often arrays or collections). LINQ is a **very powerful tool**, similar to most SQL languages by logic and syntax. It actually works with collections in the same way as SQL languages work with table rows in databases. It is part of the syntax of C# and Visual Basic .NET and consists of few special keywords like **from**, **in** and **select**. In order to use LINQ queries in C#, we have to include a **reference to System.Core.dll** and to **include the namespace System.Linq** in the beginning of the C# program.

Data Sources with LINQ

To define the data source (collection, array and so on), we have to use the keywords **from** and **in** and a variable for the iteration of the collection (the iteration is similar to the one with the **foreach** operator). For example, a query that starts like this:

```
from culture
in CultureInfo.GetCultures(CultureTypes.AllCultures)
```

can be read as follows: "for each element of the collection **CultureInfo.GetCultures(CultureTypes.AllCultures)** assign the variable **culture** and use it to refer to these items further in the query".

Data Filtering with LINQ

The keyword **where** can be used to set conditions, that should be kept by each item of the collection, in order to continue with the execution of the query. The expression after **where** is always of a Boolean type. We can say that **where works as a filter for the elements**. For example, if we want to see only those cultures, whose name begins with the lowercase Latin letter **b**, we can continue the query from our last example like this:

```
where culture.Name.StartsWith("b")
```

As we can notice, after **where … in**, we use only the name we gave for the iteration of the variables in the collection. The keyword **where** is compiled up to the invoking of the extension method **Where()**.

```
where culture.Name.StartsWith("b")
```

Results of LINQ Queries

To **choose the output data** for the query, we can **use the keyword select**. The result is an object of an existing class or an anonymous type. The result can also be a property of the objects, the query runs through or the objects themselves. The **select** statement and everything following it is placed **always at the end of the query**. The four keywords: **from**, **in**, **where** and **select**, are completely enough to create a simple LINQ query. Here is an example:

```
List<int> numbers = new List<int>() {
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};
var evenNumbers =
   from num in numbers
   where num % 2 == 0
   select num;
foreach (var item in evenNumbers)
{
   Console.Write(item + " ");
}
```

The result is:

```
2 4 6 8 10
```

The example above runs a **query over a collection of integers called numbers** and filters only the even ones in a new collection. The query can be read as follows: "for each number **num** from **numbers** check if it is multiple of 2, and if so, add it to the new collection".

Sorting Data with LINQ

Sorting with LINQ queries is done through the **keyword orderby**. The conditions, used for sorting the elements, are placed after it. For each condition the order of arrangement can be indicated: ascending (using the keyword **ascending**) and descending (with the keyword **descending**), as by default the elements are ordered in ascending order. If we want to sort an array of strings by their length in descending order, for example, we can write the following query:

```
string[] words = { "cherry", "apple", "blueberry" };
var wordsSortedByLength =
   from word in words
   orderby word.Length descending
   select word;
foreach (var word in wordsSortedByLength)
{
   Console.WriteLine(word);
}
```

The result is:

```
blueberry
cherry
apple
```

If no instruction for the order is given (i.e. the keyword **orderby** is missing from the query) the items are printed in the way they would be processed, if the **foreach** operator was used.

Grouping Results with LINQ

To group the results by some criteria the keyword **group** should be used. The pattern is as follows:

```
group [variable name] by [grouping condition] into [group name]
```

The **result of grouping is a new collection of a special type** that can be used further in the query. After the grouping, however, the query stops working with its initial variable. This means that in the **select** statement, we can use only the group. An example of grouping:

```
int[] numbers =
   { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0, 10, 11, 12, 13 };
int divisor = 5;

var numberGroups =
   from number in numbers
   group number by number % divisor into group
   select new { Remainder = group.Key, Numbers = group };

foreach (var group in numberGroups)
{
   Console.WriteLine(
   "Numbers with a remainder of {0} when divided by {1}:",
      group.Remainder, divisor);
   foreach (var number in group.Numbers)
   {
      Console.WriteLine(number);
   }
}
```

The result is:

```
Numbers with a remainder of 0 when divided by 5:
5
0
10
Numbers with a remainder of 4 when divided by 5:
4
9
Numbers with a remainder of 1 when divided by 5:
1
6
11
Numbers with a remainder of 3 when divided by 5:
3
8
13
Numbers with a remainder of 2 when divided by 5:
7
2
12
```

As we can see from the example above, the numbers printed to the console are grouped by their remainders of the division by 5. In the query, for each number **`number % divisor`** is calculated, and for each different result a new

group is formed. Further in the query, the **select** operator works on the list of created groups, and for each group creates an anonymous type with two properties: **Remainder** and **Numbers**. To the property **Remainder** the **key of the group** is assigned (in our case the remainder of the division by the **divisor** of the number). And to the property **Numbers** the collection **group** is assigned, that contains all the elements in the group. Notice that **select** is executed only over the list of groups. The variable **number** cannot be used there. Further in the example of two nested **foreach** statements, the remainders (the groups) and the numbers that have the remainder (located in the group) are printed.

Joining Data with LINQ

The **join** statement is a bit more complicated than the other LINQ statements. It joins collections by certain matching criteria and extracts the needed data. Its syntax is as follows:

```
from [variable name from collection 1] in [collection 1] join
[variable name from collection 2] in [collection 2] on [part of
the compare condition from collection 1] equals [part of the
compare condition from collection 2]
```

Further in the query (e.g. in the **select** part), both, the name of the variable from collection 1, and the name of the variable from collection 2, can be used. Example:

```
public class Product
{
   public string Name { get; set; }
   public int CategoryID { get; set; }
}

public class Category
{
   public int ID { get; set; }
   public string Name { get; set; }
}
```

The code that illustrates how to use LINQ joins:

```
List<Category> categories = new List<Category>()
{
   new Category() { ID = 1, Name = "Fruit" },
   new Category() { ID = 2, Name = "Food" },
   new Category() { ID = 3, Name = "Shoe" },
   new Category() { ID = 4, Name = "Juice" },
```

```
};
List<Product> products = new List<Product>()
{
   new Product() { Name = "Strawberry", CategoryID = 1 },
   new Product() { Name = "Banana", CategoryID = 1 },
   new Product() { Name = "Chicken meat", CategoryID = 2 },
   new Product() { Name = "Apple Juice", CategoryID = 4 },
   new Product() { Name = "Fish", CategoryID = 2 },
   new Product() { Name = "Orange Juice", CategoryID = 4 },
   new Product() { Name = "Sandal", CategoryID = 3 },
};
var productsWithCategories =
   from product in products
   join category in categories
     on product.CategoryID equals category.ID
   select new {  Name = product.Name,
                 Category = category.Name };
foreach (var item in productsWithCategories)
{
   Console.WriteLine(item);
}
```

The result is:

```
{ Name = Strawberry, Category = Fruit }
{ Name = Banana, Category = Fruit }
{ Name = Chicken meat, Category = Food }
{ Name = Apple Juice, Category = Juice }
{ Name = Fish, Category = Food }
{ Name = Orange Juice, Category = Juice }
{ Name = Sandal, Category = Shoe }
```

In the example above, we create two classes and an imaginary relationship between them. To each product some category **CategoryID** (represented by a number) corresponds, that matches the number **ID** from the class **Category** in the collection **categories**. If we want to use this relation and to create a new anonymous type, where to store the products and their names and category, we can write the above LINQ query. It joins the collection of elements of type **Category** with the one of type **Product** by the mentioned criteria (match between **ID** from **Category** and **CategoryID** from **Products**). In the **select** part of the query, we use both names **category** and **product** to construct an anonymous type with the name of the product and the name of the category.

Nested LINQ Queries

LINQ also supports **nested queries**. For example our last query can be written by nesting two queries in the following way (the result is exactly the same as the one with **join**):

```
var productsWithCategories =
  from product in products
  select new {
    Name = product.Name,
    Category =
      (from category in categories
      where category.ID == product.CategoryID
      select category.Name).First()
  };
```

Since each query in LINQ returns a collection of items (irrespective of whether the result from it is of 0, 1 or more elements), we need to use the extension method **First()** over the result of the nested query. The method **First()** returns the first element (in our case the only one) of the collection it is applied on. In this way we get the name of the category only by its ID number.

LINQ Performance

As a rule using **LINQ and extension methods is slower than using direct operations** over a collection of elements, so beware of using LINQ when processing large collections or the performance is critical.

Let's compare the speed of adding 50,000,000 elements to a list through extension methods and directly with a **for**-loop:

```
List<int> l1 = new List<int>();
DateTime startTime = DateTime.Now;
l1.AddRange(Enumerable.Range(1, 50000000));
Console.WriteLine("Ext.method:\t{0}", DateTime.Now - startTime);

startTime = DateTime.Now;
List<int> l2 = new List<int>();
for (int i = 0; i < 50000000; i++) l2.Add(i);
Console.WriteLine("For-loop:\t{0}", DateTime.Now - startTime);
```

The result might be as follows (depends on the computer's CPU speed):

```
Ext.method:     00:00:01.6430939
For-loop:       00:00:00.9120522
```

LINQ technology and extension methods work through the concept of **expression trees**. Each LINQ query is translated by the compiler to an expression tree and is executed when its results are actually accessed (not earlier). For example let's consider the following code:

```
List<int> list = new List<int>();
list.AddRange(Enumerable.Range(1, 100000));

DateTime start = DateTime.Now;
for (int i = 0; i < 10000; i++)
{
   var elements = list.Where(e => e > 20000);
}
Console.WriteLine("No execution:\t{0}", DateTime.Now - start);

start = DateTime.Now;
for (int i = 0; i < 10000; i++)
{
   var element = list.Where(e => e > 20000).First();
}
Console.WriteLine("Execution:\t{0}", DateTime.Now - start);
```

The result might be as follows (depends on the computer's CPU speed):

```
No execution:   00:00:00.0070004
Execution:      00:00:02.7231558
```

This shows that if we call a **.Where(…)** filter (or **where** clause in LINQ) it is not actually executed until its result is actually needed. The elements **get filtered on demand**, at the time they are really required. In our case this is when we invoke **First()** method. Moreover, if we get the first element of a sequence, the rest elements are not processes until needed. Thus if we use change the filtering lambda function from "**e => e > 20000**" to "**e => e > 500000**", the filtering becomes times slower because more elements are processed until the first matching the filtering condition is found:

```
No execution:   00:00:00.0060004
Execution:      00:00:06.3663641
```

Standard .NET Framework collection classes like **List<T>**, **HashSet<T>** and **Dictionary<K,V>** are optimized to work fast with LINQ. Most operations with LINQ work almost as fast as if we run them directly. Let's check this example:

```
HashSet<Guid> set = new HashSet<Guid>();
for (int i = 0; i < 50000; i++)
{
```

```
   set.Add(Guid.NewGuid()); // Add random GUID
}

Guid keyForSearching = new Guid();
DateTime start = DateTime.Now;
for (int i = 0; i < 50000; i++)
{
   // Use HashSet.Contains(…)
   bool found = set.Contains(keyForSearching);
}
Console.WriteLine("HashSet: {0}", DateTime.Now - start);

start = DateTime.Now;
for (int i = 0; i < 50000; i++)
{
   // Use IEnumerable<Guid>.Contains(…) extension method
   bool found = set.Contains<Guid>(keyForSearching);
}
Console.WriteLine("Contains: {0}", DateTime.Now - start);

start = DateTime.Now;
for (int i = 0; i < 50000; i++)
{
   // Use IEnumerable<Guid>.Where(…) extension method
   bool found = set.Where(g => g==keyForSearching).Count() > 0;
}
Console.WriteLine("Where: {0}", DateTime.Now - start);
```

The result is as follows (though it depends on the computer's CPU speed):

```
HashSet: 00:00:00.0030002
Contains: 00:00:00.0040003
Where: 00:02:49.9717218
```

Seems like .NET Framework takes into account the capability to search in constant time O(1) in a **HashSet<T>**, so searching though the native method **Contains(…)** and though the extension methods **IEnumerable.Contains(…)** both **run in time O(1)**. By contrast, the **IEnumerable.Where(…)** method is **dramatically slower** and runs in linear time O(n). This is expected, because the **Where(…)** method checks certain condition for each element in a collection and it is expected to process all elements one by one. By contrast the **Contains(…)** method just searches for single element which is fast operation.

In case you do not remember about the asymptotic notation O(1) and O(n), please check the chapter "Data Structures and Algorithm Complexity".

In the above example we use the system structure **Guid**. This is a global unique identifier often used in computer technologies to identify an object. It may look like the following: **8668f585-faf8-4685-8025-6a8d1d2aba0a**. If you want to generate a global unique (world-wide) identifier, you might benefit from the method **Guid.NewGuid()**, like we do in the code above.

Exercises

1. Implement an extension method **Substring(int index, int length)** for the class **StringBuilder** that returns a new **StringBuilder** and has the same functionality as the method **Substring(…)** of the class **String**.
2. Implement the following extension methods for the classes, implementing the interface **IEnumerable<T>**: **Sum()**, **Min()**, **Max()**, **Average()**.
3. Write a class **Student** with the following properties: first name, last name and age. Write a method that for a given array of students finds those, whose first name is before their last one in alphabetical order. Use LINQ.
4. Create a **LINQ query** that finds the first and the last name of all students, aged between 18 and 24 years including. Use the class **Student** from the previous exercise.
5. By using the extension methods **OrderBy(…)** and **ThenBy(…)** with **lambda expression**, **sort a list of students** by their first and last name in descending order. Rewrite the same functionality using a **LINQ query**.
6. Write a program that prints to the console all numbers from a given array (or list), that are **multiples of 7 and 3 at the same time**. Use the built-in **extension methods** with **lambda expressions** and then rewrite the same using a **LINQ query**.
7. Write an extension method for the class **String** that **capitalizes all letters**, which are the beginning of a word in a sentence in English. For example: "**this iS a Sample sentence.**" should be converted to "**This Is A Sample Sentence.**".
8. Create a hash-table to hold a **phone book**: a set of person names and their phone numbers (e.g. Kate Wilson → +3592981981, +3598862536; Alex & Co. → 1-800-ALEX; Steve Milton → +496023456). Fill enough random data (e.g. 50,000 key-value pairs). Measure **how much time it takes** to perform searching by key in the hash-table using its native search capabilities, using the extension methods **IEnumerable.Contains(…)** and **IEnumerable.Where(…)**. Can you explain the difference?

Solutions and Guidelines

1. Follow the syntax explained in the **section "Extension Methods"**. You may create a new **StringBuilder** and to write in it all the characters with indices, starting from **index** and with length **length**, from the object that the extension method will work on.

2. For generic implementation of the **Min()** and **Max()** methods for any generic type **T** you can add a restriction to the passed type **T** to be **comparable**, i.e. you should have something like this:

```
public static T Min<T>(this IEnumerable<T> elements)
  where T : IComparable<T>
{ … }
```

   Since not all data types have predefined operators **+** and **/**, it will not be possible to apply the functions **Sum()** and **Average()** to all types directly. There are no interfaces **ISummable<T>** and **IDividable<T>** in .NET. One way to work around this problem is to **convert all input objects to decimal** and then to calculate sum / average and return **decimal** as result. For the conversion you can use the static method **Convert.ToDecimal(…)**.

   Another interesting approach is to use the **dynamic data type** in C# to hold the arguments and results and to execute the operations over them **at runtime** (due to the dynamic evaluation capabilities in C#):

```
public static dynamic Min<T>(this IEnumerable<T> elements)
{ … }
```

   This is **easier to implement** and works better but could have performance issues and some special cases to be handled.

3. Review the keywords **from**, **where** and **select** from the "LINQ Queries" section in this chapter.

4. Write a **LINQ query** to select the described students in an **anonymous type** that contains only two properties – **FirstName** and **LastName**.

5. For the **LINQ query** use **from**, **orderby**, **descending** and **select**. For the implementation with the **lambda expressions**, you can use the methods **OrderByDescending(…)** and **ThenByDescending(…)**.

6. It is enough to check if the numbers are multiples of 21, instead of writing two **where** conditions.

7. Use the method **ToTitleCase(…)** of the property **TextInfo** in the culture **en-US** in the following way:

```
new CultureInfo("en-US", false).TextInfo.ToTitleCase(text);
```

8. See the examples at the end of the section "LINQ Performance". You can use **Dictionary<string, List<string>>** to hold the phone book. You may **explain the difference in the execution speed** by trying to explain how searching works internally and by the assumption that searching in a hash-table takes time O(1) and searching in a collection element by element runs in linear time O(n).