44. Collections - History and Overview

This chapter is the first in our coverage of collections.

Collections are used to organize and process a number of objects or values of the same type. In almost any real-life program, collections of objects or values play important roles.

Collections fit nicely in our agenda of object-oriented programming. A collection holds a number of objects (of the same type), but a concrete collection is also itself an object. The commonalities of a number of collections objects are described by the type of the collection objects. In the following chapters we will encounter a number of different interfaces and classes, which represent collection types. Not surprisingly, generic types as discussed in Chapter 42, play an important role when we wish to deal with collections that are constrained to contain only objects of a particular element type.

In the rest of this short introductory chapter we will briefly outline the historic development of collection programming. In the main part of the lecture, Chapter 45 and Chapter 46, we deal with two main categories of collections: Lists and Dictionaries.

44.1. A historic View on Collection Programming

Lecture 12 - slide 2

We identify three stages or epochs related to the development of collections:

- Native arrays and custom made lists
    - *Fixed sized arrays* - limited set of operations
    - *Variable sized linked lists* - direct pointer manipulation
- First generation collection classes
    - Elements of type `Object` - Flexible sizing - Rich repertoire of operations
    - Type unsafe - Casting - Inhomogeneous collections
- Second generation collection classes
    - The flexibility of the first generation collections remains
    - Type safe - Generic - Type parameterized - Homogeneous

Arrays are fundamental in imperative programming, for instance in C. In older programs - or old-fashioned programs - many collections are dealt with by means of arrays. Many modern programs still use arrays for collections, either due to old habits or because of the inherent efficiency of array processing. The efficiency of arrays stems from the fact that the memory needed for the elements is allocated as a single consecutive area of fixed size.

Another fundamental technique for dealing with collections is encountered in linked lists. In linked list one elements is connected to the next element by a pointer. The linking is done by use of pointers. In single-linked list, an element is linked to its successor. In double-linked list, an element is both linked to its successor and to its predecessor. Linked trees, such as binary trees, are also common. In some languages (such as C and Pascal) linked data structures require explicit pointer manipulation. Other languages (such as Lisp) hide the pointers behind the scene.

First generation collection classes deemphasize the concrete representation of collections. Instead, the capabilities and interfaces (such as insertion, deletion, searching, conversion, etc) of collections are brought into focus. This reflects good and solid object-oriented thinking. Typical first-generation collection classes blur the distinction between (consecutive) arrays and (linked) lists. The concept of an `ArrayList` is seen both in early versions of Java and C#. Collection concepts are organized in type hierarchies: A `List` ***is a*** `Collection` and a `Set` ***is a*** `Collection` (see Section 25.2). The element type of collections is the most general type in the system, namely `Object`. As a consequence of this, it is hard to avoid collection of "pears" and "bananas" (inhomogeneous collections). Thus, type safeness must be dealt with at run-time. This is against the trend of static type checking and type safety. We will briefly review the first generation collection classes of C# in Chapter 47.

The second (and current) generation of collections make use of generic types (type parameterized classes and interfaces), as discussed in Chapter 42. The weaknesses of the first generation collection classes have been the primary motivation for introduction all the complexity of genericity (see Chapter 41 where we motivated generic classes by a study of the class `Set`). With use of type parameterized classes we can statically express **`List<Banana>`** and **`List<Pear>`** and hereby eliminate the risk of type errors at run time. In the following chapters we will - with the exception of Chapter 47 - limit ourselves to study type parameterized collections.

45. Generic Collections in C#

In this chapter we will study different list interfaces and classes.

45.1. Overview of Generic Collections in C#

Lecture 12 - slide 4

We start by showing a type hierarchy of list-related types. The white boxes in Figure 45.1 are interfaces and the grey boxes are classes.

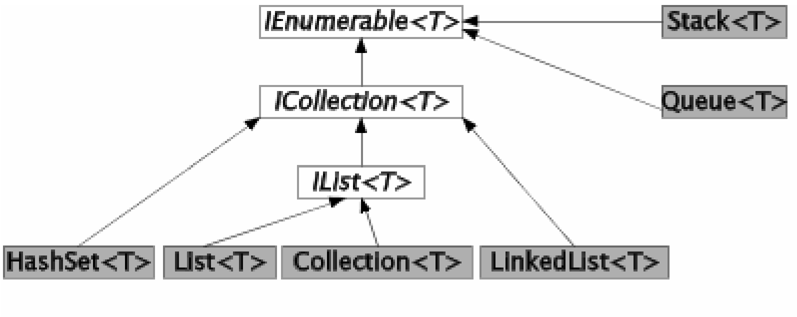

Figure 45.1 *The class and interface inheritance tree related to Lists*

All interfaces and classes seen in Figure 45.1, apart from **`Stack<T>`** and **`Queue<T>`**, will be discussed in the forthcoming sections of the current chapter.

The class `System.Array` (see Section 28.2) which conceptually is the superclass of all native array types in C#, also implements the generic interfaces `IList<T>`. Notice, however, that `Array` 's implementation of `IList<T>` is carried out by special means, and that it does not show up in the usual C# documentation. A more detailed discussion of the `Array` class is carried out in Section 47.1.

Version 3.5 of the .NET Framework contains a class, `HashSet<T>`, that supports the mathematical set concept. As such, it is similar to the class `Set<T>`, which we used as example for introduction of generic types in Section 42.1. `HashSet<T>` is, however, much more efficient than `Set<T>`.

45.2. The Interface IEnumerable<T>

Lecture 12 - slide 5

At the most general level of Figure 45.1 *traversability* is emphasized. This covers the ability to step through all elements of a collection. The interface `IEnumerable<T>` announces one parameterless method called `GetEnumerator`. The type parameter `T` is the type of the elements in the collection.

- Operations in the interface **`IEnumerable<T>`**:
    - `IEnumerator<T>` **`GetEnumerator`** ( )

As the name indicates, `GetEnumerator` returns an enumerator, which offers the following interface:

- Operations in the interface **`IEnumerator<T>`**:
    - `T` **`Current`**
    - `bool` **`MoveNext`**( )
    - `void` **`Reset`** ( )

We have discussed the non-generic versions of both interfaces in Section 31.6. An `IEnumerator` object is used as the basis of traversal in a **`foreach`** loop.

Without access to an `IEnumerator` object it would not be possible to traverse the elements of a collection in a **foreach** loop. You do not very often use the `GetEnumerator` operation explicitly in your own program, but you most probably rely on it implicitly! The reason is that many of your collections are traversed, from one end to the other, by use of **foreach**. The **foreach** control structure would not work without the operation `GetEnumerator`. As you can see from Figure 45.1 all of our collections implement the interface `IEnumerable<T>` and hereby they provide the operation `GetEnumerator`.

It is worth noticing that an object of type `IEnumerator<T>` does not support removal of elements from the collection. In C# it is therefore not allowed to remove elements during traversal of a collection in a **foreach** loop. In the Java counterpart to `IEnumerator<T>` (called `Iterator` in Java), there is a `remove` method. The `remove` method can be called once for each step forward in the collection. `remove` is an optional operation in the Java `Iterator` interface. Consequently, removal of elements is not necessarily supported by all implementations of the Java `Iterator` interface.

45.3. The Interface ICollection<T>

Lecture 12 - slide 6

At the next level of Figure 45.1 we encounter the `ICollection<T>` interface. It can be summarized as follows.

- Operations in the interface **`ICollection<T>`**:
    - *The operation prescribed in the superinterface* **`IEnumerable<T>`**
    - bool **`Contains`**(**`T`** element)
    - void **`Add`**(**`T`** element)
    - bool **`Remove`**(**`T`** element)
    - void **`Clear`**()
    - void **`CopyTo`**(**`T[]`** targetArray, int startIndex)
    - int **`Count`**
    - bool **`IsReadOnly`**

In addition to traversability, elements of type `T` can be added to and removed from objects of type `ICollection<T>`. At this level of abstraction, it is not specified where in the collection an element is added. As listed about, a few other operations are supported: Membership testing (`Contains`), resetting (`Clear`), copying of the collection to an array (`CopyTo`), and measuring of size (`Count`). Some collections cannot be mutated once they have been created. The IsReadOnly property allows us to find out if a given ICollection object is a read only collection.

45.4. The Interface IList<T>

Lecture 12 - slide 7

At the next level of interfaces in Figure 45.1 we meet IList<T>. This interface prescribes random access to elements.

- Operations in the interface **IList<T>**:
    - *Those prescribed in the superinterfaces* **ICollection<T>** *and* **IEnumerable<T>**
    - **T this**[int index]
    - int **IndexOf**(**T** element)
    - void **Insert**(int index, **T** element)
    - void **RemoveAt**(int index)

In addition to ICollection<T>, the type IList<T> allows for indexed access to the T elements. The first mentioned operation (this) is an indexer, and IndexOf is its inverse operation. (See Chapter 19 for a general discussion of indexers). In addition, IList<T> has operations for inserting and removing elements at given index positions.

45.5. Overview of the class Collection<T>

Lecture 12 - slide 8

We now encounter the first class in the collection hierarchy, namely Collection<T>. Most interfaces and classes discussed in this chapter belong to the namespace System.Collections.Generic, but of some odd reason the class Collection<T> belongs to System.Collections.ObjectModel.

As can be seen from Figure 45.1 the generic class Collection<T> implements the generic interface IList<T>. As such it supports all the operations of the three interfaces we discussed in Section 45.2 - Section 45.4. As it appears from Figure 45.1 the generic class List<T> implements the same interface. It turns out that Collection<T> is a minimal class which implements the three interfaces, and not much more. As we will see in Section 45.9, List<T> has many more operations, most of which are not prescribed by the interfaces it implement.

Basically, an instance of Collection<T> supports indexed access to its elements. Contrary to arrays, however, there is no limit on the number of elements in the collection. The generic class Collection<T> has another twist: It is well suited as a superclass for specialized (non-generic) collections. We will see why and how in Section 45.7.

We will not summarize the public interface of Collection<T> in the paper version of material, because it is the sum of the interfaces of IEnumerable<T>, ICollection<T>, and IList<T>. You should, however notice the two constructors of Collection<T>, a parameterless constructor and a non-copying, "wrapping" constructor on an IList<T>.

Collection initializers are new in C# 3.0. Instead of initializing a collection via an IList, typically an array, such as in

```
Collection<int> lst = new Collection<int>(new int[]{1, 2, 3, 4});
```

it is possible in C# 3.0 to make use of collection initializers:

```
Collection<int> lst = new Collection{1, 2, 3, 4};
```

A collection initializer uses the Add method repeatedly to insert the elements within **{...}** into an empty list.

Collection initializers are often used in concert with *object initializers*, see Section 18.4, to provide for smooth creation of collection of objects, which are instances of our own types.

You may be interested to know details of the actual representation (data structure) used internally in the generic class Collection<T>. Is it an array? Is it a linked list? Or is it something else, such as a mix of arrays and lists, or a tree structure? Most likely, it is a resizeable array. Notice however that from an object-oriented programming point of view (implying encapsulation and visibility control) it is inappropriate to ask such a question. It is sufficient to know about the interface of Collection<T> together with the time complexities of the involved operations. (As an additional remark, the source code of the C# libraries written by Microsoft is not generally available for inspection. Therefore we cannot easily check the representation details of the class). The interface of Collection<T> includes details about the execution times of the operations of Collection<T> relative to the size of a collection. We deal with timing issues of the operations in the collection classes in Section 45.17.

45.6. Sample use of class Collection<T>

Lecture 12 - slide 9

Let us now write a program that shows how to use the central operations in Collection<T>. In Program 45.1 we use an instance of the constructed class Collection<char>. Thus, we deal with a collection of character values. It is actually worth noticing that we in C# can deal with collections of value types (such as Collection<char>) as well as collections of reference types (such as Collection<Point>).

```
using System;
using System.Collections.ObjectModel;
using System.Collections.Generic;

class BasicCollectionDemo{

  public static void Main(){

    // Initialization - use of a collection initializer. After that add 2 elements.
    IList<char> lst = new Collection<char>{'a', 'b', 'c'};
    lst.Add('d'); lst.Add('e');
    ReportList("Initial List", lst);

    // Mutate existing elements in the list:
    lst[0] = 'z'; lst[1]++;
    ReportList("lst[0] = 'z'; lst[1]++;", lst);

    // Insert and push towards the end:
```

```
19     lst.Insert(0,'n');
20     ReportList("lst.Insert(0,'n');", lst);
21 
22     // Insert at end - with Insert:
23     lst.Insert(lst.Count,'x');      // equivalent to lst.Add('x');
24     ReportList("lst.Insert(lst.Count,'x');", lst);
25 
26     // Remove element 0 and pull toward the beginning:
27     lst.RemoveAt(0);
28     ReportList("lst.RemoveAt(0);", lst);
29 
30     // Remove first occurrence of 'c':
31     lst.Remove('c');
32     ReportList("lst.Remove('c');", lst);
33 
34     // Remove remaining elements:
35     lst.Clear();
36     ReportList("lst.Clear(); ", lst);
37 
38   }
39 
40   public static void ReportList<T>(string explanation, IList<T> list){
41     Console.WriteLine(explanation);
42     foreach(T el in list)
43       Console.Write("{0, 3}", el);
44     Console.WriteLine(); Console.WriteLine();
45   }
46 
47 }
```

Program 45.1 *Basic operations on a Collection of characters.*

The program shown above explains itself in the comments, and the program output in Listing 45.2 is also relatively self-contained. Notice the use of the *collection initializer* in line 9 of Program 45.1. As mentioned in Section 45.5 collection initializers have been introduced in C# 3.0. In earlier versions of C# it was necessary to initialize a collection by use or an *array initializer* (see the discussion of Program 6.7) via the second constructor mentioned above.

```
Initial List
  a  b  c  d  e

lst[0] = 'z'; lst[1]++;
  z  c  c  d  e

lst.Insert(0,'n');
  n  z  c  c  d  e

lst.Insert(lst.Count,'x');
  n  z  c  c  d  e  x

lst.RemoveAt(0);
  z  c  c  d  e  x

lst.Remove('c');
  z  c  d  e  x

lst.Clear();
```

Listing 45.2 *Output of the program with basic operations on a Collection of characters.*

We make the following important observations about the operations in `Collection<T>`:

- The indexer `lst[idx] = expr` mutates an existing element in the collection
    - *The length of the collection is unchanged*
- The `Insert` operation splices a new element into the collection
    - Push subsequent elements towards the end of the collection
    - *Makes the collection longer*
- The `Remove` and `RemoveAt` operations take elements out of the collections
    - Pull subsequent elements towards the beginning of the collection
    - *Makes the collection shorter*

45.7. Specialization of Collections

Lecture 12 - slide 10

Let us now assume that we wish to make our own, specialized (non-generic) collection class of a particular type of objects. Below we will - for illustrative purposes - write a class called `AnimalFarm` which is intended to hold instances of class `Animal`. It is reasonable to program `AnimalFarm` as a subclass of an existing collection class. In this section we shall see that `Collection<Animal>` is a good choice of superclass of `AnimalFarm`.

The class `AnimalFarm` depends on the class `Animal`. You are invited to take a look at class `Animal` via the accompanying slide . We do not include class `Animal` here because it does not add new insight to our interests in collection classes. The four operations of class `AnimalFarm` are shown below.

```
using System;
using System.Collections.ObjectModel;

public class AnimalFarm: Collection<Animal>{

  protected override void InsertItem(int i, Animal a){
    base.InsertItem(i,a);
    Console.WriteLine("**InsertItem: {0}, {1}", i, a);
  }

  protected override void SetItem(int i, Animal a){
    base.SetItem(i,a);
    Console.WriteLine("**SetItem: {0}, {1}", i, a);
  }

  protected override void RemoveItem(int i){
    base.RemoveItem(i);
    Console.WriteLine("**RemoveItem: {0}", i);
  }

  protected override void ClearItems(){
    base.ClearItems();
    Console.WriteLine("**ClearItems");
  }

}
```

Program 45.3 *A class AnimalFarm - a subclass of* **`Collection<Animal>`** *- testing protected members.*

It is important to notice that the four highlighted operations in Program 45.3 are redefinitions of virtual, protected methods in Collection<Animal>. Each of the methods activate the similar method in the superclass (this is method combination). In addition, they reveal on standard output that the protected method has been called. A more realistic example of class AnimalFarm will be presented in Program 45.6.

The four operations are not part of the client interface of class AnimalFarm. They are protected operations. The client interface of AnimalFarm is identical to the public operations inherited from Collection<Animal>. It means that we use the operations Add, Insert, Remove etc. on instances of class AnimalFarm.

We should now understand the role of the four protected operations InsertItem, RemoveItem, SetItem, and ClearItems relative to the operations in the public client interface. Whenever an element is inserted into a collection, the protected method InsertItem is called. Both Add and Insert are programmed by use of InsertItem. Similarly, both Remove and RemoveAt are programmed by use of RemoveItem. And so on. We see that the major functionality behind the operations in Collection<T> is controlled by the four protected methods InsertItem, RemoveItem, SetItem, and ClearItems.

```
using System;
using System.Collections.ObjectModel;

class App{

  public static void Main(){

    AnimalFarm af = new AnimalFarm();

    // Populating the farm with Add
    af.Add(new Animal("elephant"));
    af.Add(new Animal("giraffe"));
    af.Add(new Animal("tiger"));
    ReportList("Adding elephant, giraffe, and tiger with Add(...)", af);

    // Additional population with Insert
    af.Insert(0, new Animal("dog"));
    af.Insert(0, new Animal("cat"));
    ReportList("Inserting dog and cat at index 0 with Insert(0, ...)", af);

    // Mutate the animal farm:
    af[1] = new Animal("herring", AnimalGroup.Fish, Sex.Male);
    ReportList("After af[1] = herring", af);

    // Remove tiger
    af.Remove(new Animal("tiger"));
    ReportList("Removing tiger with Remove(...)", af);

    // Remove animal at index 2
    af.RemoveAt(2);
    ReportList("Removing animal at index 2, with RemoveAt(2)", af);

    // Clear the farm
    af.Clear();
    ReportList("Clear the farm with Clear()", af);
  }

  public static void ReportList<T>(string explanation, Collection<T> list){
    Console.WriteLine(explanation);
    foreach(T el in list)
      Console.WriteLine("{0, 3}", el);
    Console.WriteLine(); Console.WriteLine();
  }
}
```

Program 45.4 *A sample client of AnimalFarm - revealing use of protected* **Collection<Animal>** *methods.*

Take a close look at the output of Program 45.4 in Listing 45.5. The output explains the program behavior.

```
**InsertItem: 0, Animal: elephant
**InsertItem: 1, Animal: giraffe
**InsertItem: 2, Animal: tiger
Adding elephant, giraffe, and tiger with Add(...)
Animal: elephant
Animal: giraffe
Animal: tiger


**InsertItem: 0, Animal: dog
**InsertItem: 0, Animal: cat
Inserting dog and cat at index 0 with Insert(0, ...)
Animal: cat
Animal: dog
Animal: elephant
Animal: giraffe
Animal: tiger


**SetItem: 1, Animal: herring
After af[1] = herring
Animal: cat
Animal: herring
Animal: elephant
Animal: giraffe
Animal: tiger


**RemoveItem: 4
Removing tiger with Remove(...)
Animal: cat
Animal: herring
Animal: elephant
Animal: giraffe


**RemoveItem: 2
Removing animal at index 2, with RemoveAt(2)
Animal: cat
Animal: herring
Animal: giraffe


**ClearItems
Clear the farm with Clear()
```

Listing 45.5 *Output from sample client of AnimalFarm.*

45.8. Specialization of Collections - a realistic example

Lecture 12 - slide 11

The protected methods in class `AnimalFarm`, as shown in Section 45.7, did only reveal if/when the protected methods were called by other methods. In this section we will show a more realistic example that redefines the four protected methods of `Collection<T>` in a more useful way.

In the example we program the following semantics of the insertion and removal operations of class `AnimalFarm`:

- If we add an animal, an additional animal of the opposite sex is also added.
- Any animal removal or clearing of an animal farm is rejected.

In addition, we add a `GetGroup` operation to `AnimalFarm`, which returns a collection (an sub animal farm) of all animals that belongs to a given group (such as all birds).

The class `Animal` has not been changed, and it still available via accompanying slide.

```
using System;
using System.Collections.ObjectModel;

public class AnimalFarm: Collection<Animal>{

  // Auto insert animal of opposite sex
  protected override void InsertItem(int i, Animal a){
    if(a.Sex == Sex.Male){
      base.InsertItem(i,a);
      base.InsertItem(i, new Animal(a.Name, a.Group, Sex.Female));
    } else {
      base.InsertItem(i,a);
      base.InsertItem(i,new Animal(a.Name, a.Group, Sex.Male));
    }
  }

  // Prevent removal
  protected override void RemoveItem(int i){
    Console.WriteLine("[Removal denied]");
  }

  // Prevent clearing
  protected override void ClearItems(){
    Console.WriteLine("[Clearing denied]");
  }

  // Return all male animals in a given group
  public AnimalFarm GetGroup(AnimalGroup g){
    AnimalFarm res = new AnimalFarm();
    foreach(Animal a in this)
      if (a.Group == g && a.Sex == Sex.Male) res.Add(a);
    return res;
  }

}
```

Program 45.6 *The class AnimalFarm - a subclass of* ***`Collection<Animal>`***.

Notice the way we implement the rejection in `RemoveItem` and `ClearItems`: We do not call the superclass operation.

In Program 45.7 (only on web) we show an AnimalFarm client program similar (but not not identical) to Program 45.4. The program output in Listing 45.8 (only on web) reveals the special semantics of the virtual, protected operations from Collection<T> - as redefined in Program 45.6.

45.9. Overview of the class List<T>

Lecture 12 - slide 12

We are now going to study the generic class List<T>. As it appears from Figure 45.1 both List<T> and Collection<T> implement the same interface, namely IList<T>, see Section 45.4. But as already noticed, List<T> offers many more operations than Collection<T>.

In the same style as in earlier sections, we provide an overview of the important operations of List<T>.

- Constructors
  - List(), List(IEnumerable<T>), List(int)
  - Via a *collection initializer*: new List<T> {t1, t2, ..., tn}
- Element access
  - this[int], GetRange(int, int)
- Measurement
  - Count, Capacity
- Element addition
  - Add(T), AddRange(IEnumerable<T>), Insert(int, T), InsertRange(int, IEnumerable<T>)
- Element removal
  - Remove(T), RemoveAll(Predicate<T>), RemoveAt(int), RemoveRange(int, int), Clear()
- Reorganization
  - Reverse(), Reverse(int, int), Sort(), Sort(Comparison<T>), Sort(IComparer<T>), Sort(int, int, IComparer<T>)
- Searching
  - BinarySearch(T), BinarySearch(int, int, T, IComparer<T>), BinarySearch(T, IComparer<T>)
  - Find(Predicate<T>), FindAll(Predicate<T>), FindIndex(Predicate<T>), FindLast(Predicate<T>), FindLastIndex(Predicate<T>), IndexOf(T), LastIndexOf(T)
- Boolean queries
  - Contains(T), Exists(Predicate<T>), TrueForAll(Predicate<T>)
- Conversions
  - ConvertAll<TOutput>(Converter<T,TOutput>), CopyTo(T[]),

Compared with Collection<T> the class List<T> offers sorting, searching, reversing, and conversion operations. List<T> also has a number of "range operations" which operate on a number of elements via a single operation. We also notice a number of *higher-order operations*: Operations that take a delegate value (a function) as parameter. ConvertAll is a generic method which is parameterized with the type TOutput. ConvertAll accepts a function of delegate type which converts from type T to TOutput.

45.10. Sample use of class List<T>

Lecture 12 - slide 13

In this and the following sections we will show how to use some of the operations in `List<T>`. We start with a basic example similar to Program 45.1 in which we work on a list of characters: `List<char>`. We insert a number of `char` values into a list, and we remove some values as well. The program appears in Program 45.9 and the self-explaining output can be seen in Listing 45.10 (only on web). Notice in particular how the range operations `InsertRange` (line 28) and `RemoveRange` (line 40) operate on the list.

```
1  using System;
2  using System.Collections.Generic;
3  
4  /* Very similar to our illustration of class Collection<char> */
5  class BasicListDemo{
6  
7    public static void Main(){
8  
9      // List initialization and adding elements to the end of the list:
10     List<char> lst = new List<char>{'a', 'b', 'c'};
11     lst.Add('d'); lst.Add('e');
12     ReportList("Initial List", lst);
13 
14     // Mutate existing elements in the list
15     lst[0] = 'z'; lst[1]++;
16     ReportList("lst[0] = 'z'; lst[1]++;", lst);
17 
18     // Insert and push towards the end
19     lst.Insert(0,'n');
20     ReportList("lst.Insert(0,'n');", lst);
21 
22     // Insert at end - with Insert
23     lst.Insert(lst.Count,'x');      // equivalent to lst.Add('x');
24     ReportList("lst.Insert(lst.Count,'x');", lst);
25 
26     // Insert a new list into existing list, at position 2.
27     lst.InsertRange(2, new List<char>{'1', '2', '3', '4'});
28     ReportList("lst.InsertRange(2, new List<char>{'1', '2', '3', '4'});", lst);
29 
30     // Remove element 0 and push toward the beginning
31     lst.RemoveAt(0);
32     ReportList("lst.RemoveAt(0);", lst);
33 
34     // Remove first occurrence of 'c'
35     lst.Remove('c');
36     ReportList("lst.Remove('c');", lst);
37 
38     // Remove 2 elements, starting at element 1
39     lst.RemoveRange(1, 2);
40     ReportList("lst.RemoveRange(1, 2);", lst);
41 
42     // Remove all remaining digits
43     lst.RemoveAll(delegate(char ch){return Char.IsDigit(ch);});
44     ReportList("lst.RemoveAll(delegate(char ch){return Char.IsDigit(ch);});", lst);
45 
46     // Test of all remaining characters are letters
47     if (lst.TrueForAll(delegate(char ch){return Char.IsLetter(ch);}))
48       Console.WriteLine("All characters in lst are letters");
49     else
50       Console.WriteLine("NOT All characters in lst are letters");
51   }
```

```

  public static void ReportList<T>(string explanation, List<T> list){
    Console.WriteLine(explanation);
    foreach(T el in list)
      Console.Write("{0, 3}", el);
    Console.WriteLine(); Console.WriteLine();
  }

}
```

Program 45.9 *Basic operations on a List of characters.*

45.11. Sample use of the Find operations in List<T>

Lecture 12 - slide 14

In this section we will illustrate how to use the search operations in `List<T>`. More specifically, we will apply the methods `Find`, `FindAll` and `IndexOf` on an instance of `List<Point>`, where `Point` is a type, such as defined by the struct in Program 14.12. The operations discussed in this section do all use linear search. It means that they work by looking at one element after the other, in a rather trivial way. As a contrast, we will look at binary search operations in Section 45.13, which searches in a "more advanced" way.

In the program below - Program 45.11 - we declare a `List<Point>` in line 11, and we add six points to the list in line 13-16. In line 20 we shown how to use `Find` to locate the first point in the list whose x-coordinate is equal to 5. The same is shown in line 25. The difference between the two uses of `Find` is that the first relies on a delegate given on the fly: `delegate(Point q){return (q.Getx() == 5);}`, while the other relies on an existing static method `FindX5` (defined in line 40 - 42). The approach shown in line 20 is, in my opinion, superior.

In line 29 we show how to use the variant `FindAll`, which returns a `Point` list instead of just a single `Point`, as returned by `Find`. In line 36 we show how `IndexOf` can be used to find the index of a given `Point` in a `Point` list. It is worth asking how the `Point` parameter of `IndexOf` is compared with the points in `Point` list. The documentation states that the points are compared by use of the default equality comparer of the type `T`, which in our case is struct `Point`. We have discussed *the default equality comparer* in Section 42.9 in the slipstream of our coverage of the generic interfaces `IEquatable<T>` and `IEqualityComparer<T>`.

We use the static method `ReportList` to show a `Point` list on standard output. We call `ReportList` several times in Program 45.11. The program output is shown in Listing 45.12.

```
1  using System;
2  using System.Collections.Generic;
3
4  class C{
5
6    public static void Main(){
7
8       System.Threading.Thread.CurrentThread.CurrentCulture =
9          new System.Globalization.CultureInfo("en-US");
10
11      List<Point> pointLst = new List<Point>();
12
13      // Construct points and point list:
14      pointLst.Add(new Point(0,0)); pointLst.Add(new Point(5, 9));
15      pointLst.Add(new Point(5,4)); pointLst.Add(new Point(7.1,-13));
16      pointLst.Add(new Point(5,-2)); pointLst.Add(new Point(14,-3.4));
```

```
    ReportList("Initial point list", pointLst);

    // Find first point in list with x coordinate 5
    Point p = pointLst.Find(delegate(Point q){return (q.Getx() == 5);});
    Console.WriteLine("Found with delegate predicate: {0}\n", p);

    // Equivalent. Use predicate which is a static method
    p = pointLst.Find(new Predicate<Point>(FindX5));
    Console.WriteLine("Found with static member predicate: {0}\n", p);

    // Find all points in list with x coordinate 5
    List<Point> resLst = new List<Point>();
    resLst = pointLst.FindAll(delegate(Point q){return (q.Getx() == 5);});
    ReportList("All points with x coordinate 5", resLst);

    // Find index of a given point in pointLst.
    // Notice that Point happens to be a struct - thus value comparison
    Point searchPoint = new Point(5,4);
    Console.WriteLine("Index of {0} {1}", searchPoint,
                       pointLst.IndexOf(searchPoint));

  }

  public static bool FindX5(Point p){
    return p.Getx() == 5;
  }

  public static void ReportList<T>(string explanation,List<T> list){
    Console.WriteLine(explanation);
    int cnt = 0;
    foreach(T el in list){
      Console.Write("{0, 3}", el);
      cnt++;
      if (cnt%4 == 0) Console.WriteLine();
    }
    if (cnt%4 != 0) Console.WriteLine();
    Console.WriteLine();
  }
}
```

Program 45.11 *Sample uses of List.Find.*

```
Initial point list
Point:(0,0). Point:(5,9). Point:(5,4). Point:(7.1,-13).
Point:(5,-2). Point:(14,-3.4).

Found with delegate predicate: Point:(5,9).

Found with static member predicate: Point:(5,9).

All points with x coordinate 5
Point:(5,9). Point:(5,4). Point:(5,-2).

Index of Point:(5,4).  2
```

Listing 45.12 *Output from the Find program.*

45.12. Sample use of Sort in List<T>

Lecture 12 - slide 15

As a client user of the generic class `List<T>` it is likely that you never need to write a sorting procedure! You are supposed to use one of the already existing `Sort` methods in `List<T>`.

Sorting the elements in a collection of elements of type `T` depends on a *less than or equal operation* on `T`. If the type `T` is taken directly from the C# libraries, it may very well be the case that we can just use the default *less than or equal operation* of the type `T`. If `T` is one of our own types, we will have to supply an implementation of the comparison operation ourselves. This can be done by passing a delegate object to the `Sort` method.

Below, in Program 45.13 we illustrate most of the four overloaded `Sort` operations in `List<T>`. The actual type parameter in the example, passed for `T`, is `int`. The program output (the lists before and after sorting) is shown in Listing 45.14 (only on web).

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

class C{

  public static void Main(){

     List<int> listOriginal = new List<int>{5, 3, 2, 7, -4, 0},
               list;

     // Sorting by means of the default comparer of int:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort();
     ReportList(list);
     Console.WriteLine();

     // Equivalent - explicit notatation of the Comparer:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort(Comparer<int>.Default);
     ReportList(list);
     Console.WriteLine();

     // Equivalent - explicit instantiation of an IntComparer:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort(new IntComparer());
     ReportList(list);
     Console.WriteLine();

     // Similar - use of a delegate value for comparison:
     list = new List<int>(listOriginal);
     ReportList(list);
     list.Sort(delegate(int x, int y){
                 if (x < y)
                    return -1;
                 else if (x == y)
                    return 0;
                 else return 1;});
     ReportList(list);
     Console.WriteLine();
  }

```

```
45   public static void ReportList<T>(List<T> list){
46     foreach(T el in list)
47       Console.Write("{0, 3}", el);
48     Console.WriteLine();
49   }
50
51 }
52
53 public class IntComparer: Comparer<int>{
54   public override int Compare(int x, int y){
55     if (x < y)
56       return -1;
57     else if (x == y)
58       return 0;
59     else return 1;
60   }
61 }
```

Program 45.13 *Four different activations of the List.Sort method.*

Throughout Program 45.13 we do several sortings of `listOriginal`, as declared in line 8. In line 14 we rely the default comparer of type `int`. The default comparer is explained in the following way in the .NET framework documentation of `List.Sort`:

> This method uses the default comparer `Comparer.Default` for type `T` to determine the order of list elements. The `Comparer.Default` property checks whether type `T` implements the `IComparable` generic interface and uses that implementation, if available. If not, `Comparer.Default` checks whether type `T` implements the `IComparable` interface. If type `T` does not implement either interface, `Comparer.Default` throws an `InvalidOperationException`.

The sorting done in line 21 is equivalent to line 14. In line 21 we show how to pass *the default comparer* of type `int` explicitly to the `Sort` method.

Let us now assume the type `int` does not have a default comparer. In other words, we will have to implement the comparer ourselves. The call of `Sort` in line 28 passes a new `IntComparer` instance to `Sort`. The class `IntComparer` is programmed in line 53-61, at the bottom of Program 45.13. Notice that `IntComparer` is a subclass of `Comparer<int>`, which is an abstract class in the namespace `System.Collections.Generic`with an abstract method named `Compare`. The generic class `Comparer<T>` is in many ways similar to the class `EqualityComparer<T>`, which we touched on in Section 42.9. Most important, both have a static `Default` property, which returns a comparer object.

As a final resort that always works we can pass a comparer function to `Sort`. In C#, such a function is programmed as a delegate. (Delegates are discussed in Chapter 22). Line 35-40 shows how this can be done. Notice that the delegate we use is programmed on the fly. This style of programming is a reminiscence of *functional programming*.

I find it much more natural to pass an *ordering method* instead of *an object of a class with an ordering method*. (The latter is a left over from older object-oriented programming languages in which the only way to pass a function `F` as parameter is via an object of a class in which `F` is an instance method). In general, I also prefer to be explicit about the ordering instead of relying on some default ordering which may turn out to surprise you.

Let us summarize the lessons that we have learned from the example:

- Some types have a default comparer which is used by **`List.Sort()`**
- The default comparer of T can extracted by **`Comparer<T>.Default`**
- An *anonymous delegate comparer* is attractive if the default comparer of the type does not exist, of if it is inappropriate.

Exercise 12.1. *Shuffle List*

Write a `Shuffle` operation that disorders the elements of a collection in a random fashion. A shuffle operation is useful in many context. There is no `Shuffle` operation in **`System.Collections.Generic.List<T>`**. In the similar Java libraries there is a shuffle method.

In which class do you want to place the `Shuffle` operation? You may consider to make use of extension methods.

You can decide on programming either a mutating or a non-mutating variant of the operation. Be sure to understand the difference between these two options.

Test the Shuffle operation, for instance on **`List<Card>`**. The class `Card` (representing a playing card) is one of the classes we have seen earlier in the course.

Exercise 12.2. *Course and Project classes*

In the earlier exercise about courses and projects (found in the lecture about abstract classes and interfaces) we refined the program about `BooleanCourse`, `GradedCourse`, and `Project`. Revise your solution (or the model solution) such that the courses in the class `Project` are represented as a variable of type `List<Course>` instead of by use of four variables of type `Course`.

Reimplement and simplify the method `Passed` in class `Project`. Take advantage of the new representation of the courses in a project, such that the "3 out of 4 rule" (see the original exercise) is implemented in a more natural way.

45.13. Sample use of BinarySearch in List<T>

Lecture 12 - slide 16

The search operations discussed in Section 45.11 all implemented *linear search* processes. The search operations of this section implement *binary search* processes, which are much faster when applied on large collections. On collections of size *n*, linear search has - not surprisingly - time complexity *O(n)*. Binary search has time complexity *O(log n)*. When *n* is large, the difference between *n* and *log n* is dramatic.

The `BinarySearch` operations in `List<T>` require, as a precondition, that the list is ordered before the search is performed. If necessary, the `Sort` operation (see Section 45.12) can be used to establish the ordering.

You may ask why we should search for an element which we - in the starting point - is able to pass as input to the `BinarySearch` method. There is a couple of good answers. First, we may be interested to know if the element is present or not in the list. Second, it may also be possible to search for an incomplete object (by only comparing some selected fields in the `Comparer` method). Using this approach we are actually interested in finding the complete object, with all the data fields, in the collection.

If the `BinarySearch` operation finds an element in the list, the index of the element is returned. This is a non-negative integer. If the element is not found, a negative integer, say *i*, is returned. Below we will see that that -*i* (or more precisely the bitwise complement ~*i*) in that case is the position of the element, if it had been present in the list.

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

class BinarySearchDemo{

  public static void Main(){

     System.Threading.Thread.CurrentThread.CurrentCulture =
        new System.Globalization.CultureInfo("en-US");

     List<Point> pointLst = new List<Point>();  // Point is a struct.

     // Construct points and point list:
     pointLst.Add(new Point(0,0)); pointLst.Add(new Point(5, 9));
     pointLst.Add(new Point(5,4)); pointLst.Add(new Point(7.1,-13));
     pointLst.Add(new Point(5,-2)); pointLst.Add(new Point(14,-3.4));
     ReportList("The initial point list", pointLst);

     // Sort point list, using a specific point Comparer.
     // Notice the PointComparer:
     // Ordering according to sum of x and y coordinates
     IComparer<Point> pointComparer = new PointComparer();
     pointLst.Sort(pointComparer);
     ReportList("The sorted point list", pointLst);

     int res;
     Point searchPoint;

     // Run-time error.
     // Failed to compare two elements in the array.
//   searchPoint = new Point(5,4);
//   res = pointLst.BinarySearch(searchPoint);
//   Console.WriteLine("BinarySearch for {0}: {1}", searchPoint, res);

     searchPoint = new Point(5,4);
     res = pointLst.BinarySearch(searchPoint, pointComparer);
     Console.WriteLine("BinarySearch for {0}: {1}", searchPoint, res);

     searchPoint = new Point(1,8);
     res = pointLst.BinarySearch(searchPoint, pointComparer);
     Console.WriteLine("BinarySearch for {0}: {1}", searchPoint, res);

  }

  public static void ReportList<T>(string explanation,List<T> list){
    Console.WriteLine(explanation);
    int cnt = 0;
    foreach(T el in list){
      Console.Write("{0, 3}", el);
      cnt++;
      if (cnt%4 == 0) Console.WriteLine();
    }
    if (cnt%4 != 0) Console.WriteLine();
    Console.WriteLine();
  }

}

// Compare the sum of the x and y coordinates.
```

```
60 // Somewhat non-traditional!
61 public class PointComparer: Comparer<Point>{
62   public override int Compare(Point p1, Point p2){
63     double p1Sum = p1.Getx() + p1.Gety();
64     double p2Sum = p2.Getx() + p2.Gety();
65     if (p1Sum < p2Sum)
66       return -1;
67     else if (p1Sum == p2Sum)
68       return 0;
69     else return 1;
70   }
71 }
```

Program 45.15 *Sample uses of List.BinarySearch.*

Program 45.15 works on a list of points. Six points are created and inserted into a list in line 13-16. Next, in line 23, the list is sorted. As it appears from the `Point` comparer programmed in line 62-72, a point *p* is less than or equal to point *q*, if *p*.x + *p*.y <= *q*.x + *q*.y. You may think that this is odd, but it is our decision for this particular program example.

In line 33 we attempt to activate binary searching by use of the default comparer. But such a comparer does not exist for class *Point*. This problem is revealed at run-time.

In line 37 and 41 we search for the points (5,4) and (1,8) respectively. In both cases we expect to find the point (5,4), which happens to be located at place 3 in the sorted list. The output of the program, shown in Program 45.17 (only on web) confirms this.

In the next program, Program 45.17 we illustrate what happens if we search for a non-existing point with `BinarySearch`. The class `PointComparer` and the generic method `ReportList` are not shown in the paper version of Program 45.17. Please consult Program 45.15 where they both appear.

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

class BinarySearchDemo{

  public static void Main(){

     System.Threading.Thread.CurrentThread.CurrentCulture =
        new System.Globalization.CultureInfo("en-US");

     List<Point> pointLst = new List<Point>();

     // Construct points and point list:
     pointLst.Add(new Point(0,0)); pointLst.Add(new Point(5, 9));
     pointLst.Add(new Point(5,4)); pointLst.Add(new Point(7.1,-13));
     pointLst.Add(new Point(5,-2)); pointLst.Add(new Point(14,-3.4));
     ReportList("Initial point list", pointLst);

     // Sort point list, using a specific point Comparer:
     IComparer<Point> pointComparer = new PointComparer();
     pointLst.Sort(pointComparer);
     ReportList("Sorted point list", pointLst);

     int res;
     Point searchPoint;

     searchPoint = new Point(1,1);
     res = pointLst.BinarySearch(searchPoint, pointComparer);
     Console.WriteLine("BinarySearch for {0}: {1}\n", searchPoint, res);
```

```
30
31      if (res < 0){    // search point not found
32        pointLst.Insert(~res, searchPoint);  // Insert searchPoint such
33                                             // that pointLst remains sorted
34        Console.WriteLine("Inserting {0} at index {1}", searchPoint, ~res);
35        ReportList("Point list after insertion", pointLst);
36      }
37   }
38
39   // ReportList not shown
40 }
41
42 // Class PointComparer not shown
```

Program 45.17 *Searching for a non-existing Point.*

The scene of Program 45.17 is the same as that of Program 45.15. In line 28 we do binary searching, looking for the point (1,1). None of the points in the program have an "x plus y sum" of 2. Therefore, the point (1,1) is not located by `BinarySearch`. The `BinarySearch` method returns a negative *ghost index*. The ghost index is the bitwise complement of the index where to insert the point in such a way that the list will remain sorted. (Notice the bitwise complement operation **~** which turns 0 to 1 and 1 to 0 at the binary level). The program output reveals that position ~(-3) is the natural place of the point (1,1) to maintain the ordering of the list. Notice that the value of ~(-3) is 2, due the use of two's complement arithmetic. This explains the rationale of the negative values returned by `BinarySearch`.

The output of Program 45.17 is shown in Listing 45.18 (only on web).

Contrary to `Sort`, it is not possible to pass a delegate to `BinarySearch`. This seems to be a flaw in the design of the `List<T>` library.

We have learned the following lessons about `BinarySearch`:

- Binary search can only be done on sorted lists
- In order to use binary search, we need - in general - to provide an explicit `Comparer` object
- Binary search returns a (non-negative) integer if the element is found
    - The index of the located element
- Binary search returns a negative integer if the element is not found
    - The complement of this number is a *ghost index*
    - The index of the element if it had been in the list

45.14. Overview of the class LinkedList<T>

Lecture 12 - slide 17

The collections implemented by `Collection<T>` of Section 45.5 and `List<T>` of Section 45.9 were based on arrays. We will now turn our interest towards a list type, which is based on a *linked* representation.

Below, in Figure 45.2 we show the object-structure of a double linked list.

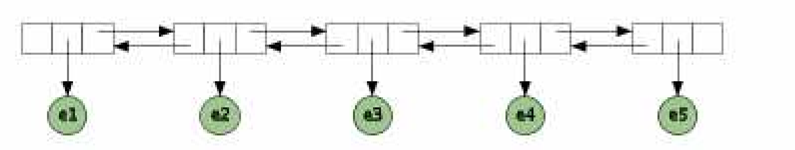

Figure 45.2 *A double linked list where instances of LinkedListNode keep the list together*

The generic class LinkedList<T> relies on a "building block class" LinkedListNode<T>. We need to deal with instances of LinkedListNodes when we work with linked lists in C#. In other words, LinkedListNode is not just a class behind the scene - it is an important class for clients of LinkedListNode<T>. In Figure 45.2 the five rectangular nodes are instances of LinkedListNode<T> for some element type T. The circular, green nodes are instances of the element type T. We will study LinkedListNode<T> in Section 45.15 after we have surveyed the list operations in LinkedList<T>.

As it can be seen from the class diagram of the list class in Figure 45.1, LinkedList<T> implements the interface ICollection<T>, see Section 45.3. Unlike Collection<T> and List<T>, LinkedList<T> does not implement indexed access, as of Ilist<T>. This is a natural choice because indexed access is not efficient in a linked representation. The following operations are available in LinkedList<T>:

- Constructors
  - LinkedList(), LinkedList(IEnumerable<T>)
- Accessors (properties)
  - First, Last, Count
- Element addition
  - AddFirst(T), AddFirst(LinkedListNode<T>), AddLast(T), AddLast(LinkedListNode<T>), AddBefore(LinkedListNode<T>, T), AddBefore(LinkedListNode<T>, LinkedListNode<T>), AddAfter(LinkedListNode<T>, T), AddAfter(LinkedListNode<T>, LinkedListNode<T>), Add(T)
- Element removal
  - Remove(T), Remove(LinkedListNode<T>), RemoveFirst(), RemoveLast(), Clear()
- Searching
  - Find(T), FindLast(T)
- Boolean queries
  - Contains(T)

A linked list can be constructed as an empty collection or as a collection filled with elements from another collection, represented as an IEnumerable<T>, see Section 45.2.

The First and Last properties access the first/last LinkedListNode in the double linked list. Count returns the number of elements in the list - not by counting them each time Count is referred - but via some bookkeeping information encapsulated in a linked list object. Thus, Count is an *O(1)* operation.

Although LinkedList<T> implements the generic interface ICollection<T>, which has a method named Add, the Add operation is not readily available on linked lists. We will in Program 45.19 show that Add is present as an explicit interface implementation, see Section 31.8. Instead of Add, the designers of LinkedList<T> want us to use one of the Add*Relative* operations: AddFirst, AddLast, AddBefore, and AddAfter. None of these are prescribed by the interface ICollection<T>, however. Each of the Add*Relative* operations are overloaded in two variants, such that we can add an element of type `T` or an object of type `LinkedListNode<T>` (which in turn reference an object of type `T`).

Using the `Remove` methods, it is possible to remove an element of type T or a specific instance of `LinkedListNode<T>`. `Remove(T)` is an *O(n)* operation; `Remove(LinkedListNode<T>)` is an *O(1)* operation. There are also parameter-less methods for removing the first/last element in the linked list. The time complexity of these are *O(1)*.

Finally there are linear search operations from either end of the list: `Find` and `FindLast`. The boolean `Contains` operation is similar to the `Find` operations. These operations all seem to rely on the `Equals` operation inherited from class `Object`. In that way `Find`, `FindLast` and `Contains` are more primitive (not as well-designed) as the similar methods in `List<T>`. (The documentation in the .NET libraries is silent about these details).

45.15. The class LinkedListNode<T>

Lecture 12 - slide 18

As illustrated in Figure 45.2, instances of the generic class `LinkedListNode<T>` keep a linked list together. In the figure, the rectangular boxes are instances of `LinkedListNode<T>`. From the figure it appears that each instance of `LinkedListNode<T>` has three references: One to the left, one to the element, and one to the right. Actually, there is a fourth reference, namely to the linked list instance to which a given `LinkedListNode` object belongs.

> The class **`LinkedListNode<T>`** is sealed, generic class that represents a non-mutable node in a linked list
>
> A `LinkedListNode` can at most belong to a single linked list

The members of `LinkedListNode<T>` are as follows:

- A single constructor **`LinkedListNode(T)`**
- Four properties
  - **`Next`** - getter
  - **`Previous`** - getter
  - **`List`** - getter
  - **`Value`** - getter and setter

The properties `Next` and `Previous` access neighbor instances of `LinkedListNode<T>`. `Value` accesses the element of type `T`. `List` accesses the linked list to which the instance of `LinkedListNode` belongs. `Next`, `Previous`, and `List` are all getters. `Value` is both a getter and a setter.

It is not possible to initialize or to mutate the fields behind the properties `Next`, `Previous`, and `List` via public interfaces. It is clearly the intention that the linked list - and only linked list - has authority to change these fields. If we programmed our own, special-purpose linked list class it would therefore not be easy to reuse the class `LinkedListNode<T>`. This is unfortunate.

Related to the discussion about the interface of `LinkedListNode<T>` we may ask how `LinkedList` is allowed to access the private/internal details of an instance of `LinkedListNode`. The best guess seems to be that the fields are internal.

45.16. Sample use of class LinkedList<T>

Lecture 12 - slide 19

We will illustrate the use of `LinkedList<T>` and `LinkedListNode<T>` in Program 45.19. In line 8 we make a linked list of integers from an array. Notice the use of the `LinkedList` constructor `LinkedList(IEnumerable<T>)`.

In line 16 we attempt to add the integer 17 to the linked list. This is not possible, because the method `Add` is not easily available, see the discussion in Section 45.14. If we insist to use `Add`, it must be done as in line 20. Most likely, you should use one of the `Add` variants instead, for instance `AddFirst` or `AddLast`.

```
1  using System;
2  using System.Collections.Generic;
3  
4  class LinkedListDemo{
5  
6    public static void Main(){
7  
8       LinkedList<int> lst = new LinkedList<int>(
9                                    new int[]{5, 3, 2, 7, -4, 0});
10 
11      ReportList("Initial LinkedList", lst);
12 
13      // Using Add.
14      // Compile-time error: 'LinkedList<int>' does not contain a
15      //                                     definition for 'Add'
16      // lst.Add(17);
17      // ReportList("lst.Add(17);" lst);
18 
19      // Add is implemented as an explicit interface implementation
20      ((ICollection<int>)lst).Add(17);
21      ReportList("((ICollection<int>)lst).Add(17);", lst);
22 
23      // Using AddFirst and AddLast
24      lst.AddFirst(-88);
25      lst.AddLast(88);
26      ReportList("lst.AddFirst(-88); lst.AddFirst(88);", lst);
27 
28      // Using Remove.
29      lst.Remove(17);
30      ReportList("lst.Remove(17);", lst);
31 
32      // Using RemoveFirst and RemoveLast
33      lst.RemoveFirst(); lst.RemoveLast();
34      ReportList("lst.RemoveFirst(); lst.RemoveLast();", lst);
35 
36      // Using Clear
37      lst.Clear();
38      ReportList("lst.Clear();", lst);
39 
40   }
41 
42   public static void ReportList<T>(string explanation, LinkedList<T> list){
43     Console.WriteLine(explanation);
```

```
    foreach(T el in list)
      Console.Write("{0, 4}", el);
    Console.WriteLine();  Console.WriteLine();
  }

}
```

Program 45.19 *Basic operations on a LinkedList of integers.*

The output of Program 45.19 is shown in Listing 45.20. By studying Listing 45.20 you will learn additional details of the `LinkedList` operations.

```
Initial LinkedList
   5   3   2   7  -4   0

((ICollection<int>)lst).Add(17);
   5   3   2   7  -4   0  17

lst.AddFirst(-88); lst.AddFirst(88);
 -88   5   3   2   7  -4   0  17  88

lst.Remove(17);
 -88   5   3   2   7  -4   0  88

lst.RemoveFirst(); lst.RemoveLast();
   5   3   2   7  -4   0

lst.Clear();
```

Listing 45.20 *Output of the program with basic operations on a LinkedList.*

The `LinkedList` example in Program 45.19 did not show how to use `LinkedListNodes` together with `LinkedList<T>`. To make up for that we will in Program 45.21 concentrate on the use of `LinkedList<T>` and `LinkedListNode<T>` together.

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

class LinkedListNodeDemo{

  public static void Main(){

     LinkedList<int> lst = new LinkedList<int>(
                              new int[]{5, 3, 2, 7, -4, 0});
     ReportList("Initial LinkedList", lst);

     LinkedListNode<int> node1, node2, node;
     node1 = lst.First;
     node2 = lst.Last;

     // Run-time error.
     // The LinkedListNode is already in the list.
     // Error message: The LinkedList node belongs a LinkedList.
/*   lst.AddLast(node1);    */

     // Move first node to last node in list
     lst.Remove(node1); lst.AddLast(node1);
     ReportList("node1 = lst.First; lst.Remove(node1); lst.AddLast(node1);", lst);

     // Navigate in list via LinkedListNode objects
```

```
26      node1 = lst.First;
27      Console.WriteLine("Third element in list: node1 = lst.First;
28 node1.Next.Next.Value    {0}\n",
29                         node1.Next.Next.Value);
30
31      // Add an integer after a LinkedListNode object
32      lst.AddAfter(node1, 17);
33      ReportList("lst.AddAfter(node1, 17);", lst);
34
35      // Add a LinkedListNode object after another LinkedListNode object
36      lst.AddAfter(node1, new LinkedListNode<int>(18));
37      ReportList("lst.AddAfter(node1, new LinkedListNode<int>(18));" , lst);
38
39      // Navigate in LinkedListNode objects and add an int before a node:
40      node = node1.Next.Next.Next;
41      lst.AddBefore(node, 99);
42      ReportList("node = node1.Next.Next.Next; lst.AddBefore(node, 99); " , lst);
43
44      // Navigate in LinkedListNode objects and remove a node.
45      node = node.Previous;
46      lst.Remove(node);
47      ReportList("node = node.Previous; lst.Remove(node);" , lst);
48
49    }
50
51    // Method ReportList not shown in this version.
   }
```

Program 45.21 *Basic operations on a LinkedList of integers - using* ***`LinkedListNode`****s.*

In line 8-9 we make the same initial integer list as in Program 45.19. In line 13-14 we see how to access to the first/last `LinkedListNode` objects of the list.

In line 19 we attempt to add `node1`, which is the first `LinkedListNode` in `lst`, as the last node of the list. This fails because it could bring the linked list into an inconsistent state. (Recall in this context that a `LinkedListNode` knows the list to which it belongs). Instead, as shown in line 22, we should first remove `node1` and then add `node1` with `AddLast`.

Please take a close look at the remaining addings, navigations, and removals in Program 45.21. As above, we show a self-explaining output of the program, see Listing 45.22.

```
Initial LinkedList
   5   3   2   7  -4   0

node1 = lst.First; lst.Remove(node1); lst.AddLast(node1);
   3   2   7  -4   0   5

Third element in list: node1 = lst.First;  node1.Next.Next.Value    7

lst.AddAfter(node1, 17);
   3  17   2   7  -4   0   5

lst.AddAfter(node1, new LinkedListNode<int>(18));
   3  18  17   2   7  -4   0   5

node = node1.Next.Next.Next; lst.AddBefore(node, 99);
   3  18  17  99   2   7  -4   0   5

node = node.Previous; lst.Remove(node);
   3  18  17   2   7  -4   0   5
```

Listing 45.22 *Output of the program with LinkedListNode operations on a LinkedList.*

45.17. Time complexity overview: Collection classes

Lecture 12 - slide 20

In this section we will discuss the efficiency of selected and important list operations in the three classes `Collection<T>`, `List<T>`, and `LinkedList<T>`. This is done by listing the *time complexities* of the operations in a table, see Table 45.1. If you are not comfortable with Big O notation, you can for instance consult Wikipedia [Big-O] or a book about algorithms and data structures.

The time complexities of the list operations are most often supplied as part of the documentation of the operations. The choice of one list type in favor of another is often based on requirements to the time complexities of important operations. Therefore you should pay careful attention to the information about time complexities in the C# library documentation.

Throughout the discussion we will assume that the lists contain *n* elements. It may be helpful to relate the table with the class diagram in Figure 45.1 from which it appears which interfaces to expect from the list classes.

Operation	`Collection<T>`	`List<T>`	`LinkedList<T>`
`this[i]`	*O(1)*	*O(1)*	-
`Count`	*O(1)*	*O(1)*	*O(1)*
`Add(e)`	*O(1) or O(n)*	*O(1) or O(n)*	*O(1)*
`Insert(i,e)`	*O(n)*	*O(n)*	-
`Remove(e)`	*O(n)*	*O(n)*	*O(n)*
`IndexOf(e)`	*O(n)*	*O(n)*	-
`Contains(e)`	*O(n)*	*O(n)*	*O(n)*
`BinarySearch(e)`	-	*O(log n)*	-
`Sort()`	-	$O(n \log n)$ or $O(n^2)$	-
`AddBefore(lln)`	-	-	*O(1)*
`AddAfter(lln,e)`	-	-	*O(1)*
`Remove(lln)`	-	-	*O(1)*
`RemoveFirst()`	-	-	*O(1)*
`RemoveLast()`	-	-	*O(1)*

Table 45.1 *Time complexities of important operations in the classes* `Collection<T>`, `List<T>`, *and* `LinkedList<T>`.

As it can be seen in the class diagram of Figure 45.1 all three classes implement the `ICollection<T>` interface with the operations `Count`, `Add`, `Remove`, and `Contains`. Thus, these four operations appear for all classes in Table 45.1.

`Count` is efficient for all lists, because it maintains an internal counter, the value of which can be returned by the `Count` property. Thus, independent of the length of a list, `Count` runs in constant time.

For all three types of lists, `Add(e)` adds an element `e` (of type `T`) to the end of the list. This can be done in constant time, because all the three types of lists have direct access the rear end of the list. The time complexity *O(1)*/*O(n)* given for `Collection<T>` and `List<T>` reflects that under normal circumstances it takes only constant time to add an element to a `Collection` or a `List`. If however, the list is full it may need resizing, and in that case the run time is linear in *n*.

`Remove(e)` and `Contains(e)`, where `e` is of type `T`, will have to search for `e` in the list. This behavior is common for all three types of lists. Therefore the run times of `Remove` and `Contains` are *O(n)*.

The indexer `this[i]` is only available in the lists that implement `Ilist<T>`. Such lists are based on arrays, and therefore the runtime of the indexer is *O(1)*. (Recall that in arrays it is possible to compute the location of an element with a given index; No searching, whatsoever, is involved).

`BinarySearch` and `Sort` are operations in `List<T>`. `Sort` implements a Quicksort variant, and as such the worst possible time complexity is $O(n^2)$, but the expected time complexity is $O(n \log n)$. The runtime of `BinarySearch` is, as expected, $O(\log n)$.

The bottom five operations in the table belong to `LinkedList`. The methods `AddBefore`, `AddAfter`, and `Remove` all work on a `LinkedListNode`, `lln`, and as such their runtimes do not depend on *n*. (Only a few references need to be assigned. The number of pointer assignments do not depend on *n*). Thus, when applied on objects of type `LinkedListNode` the runtime of these three operations are *O(1)*. `RemoveFirst` and `RemoveLast` are of time complexity *O(1)* because a linked list maintain direct references to both ends of the list.

45.18. Using Collections through Interfaces

Lecture 12 - slide 21

We started this chapter with a discussion of list interfaces, and we will end the chapter in a similar way.

It is, of course, necessary to use one of the collection classes (such as `List<T>`) when you need a collection in your program. The morale of this section is, however, that you should not use list classes more than necessary. In short, you should typically use `List<T>` or `Collection<T>` (for some type T) when you make a collection object. All other places you are better off using one of the interface types, such as `IList<T>`. The key observations can be summarized as follows.

> It is an advantage to use collections via interfaces instead of classes
>
> If possible, only use collection classes in instantiations, just after `new`
>
> This leads to programs with fewer bindings to concrete implementations of collections
>
> With this approach, it is easy to replace a collection class with another

Thus, please consider the following when you use collections:

Program against collection interfaces, not collection classes

If the types of variables and parameters are given as interfaces it is easy, a later point in time, to change the representation of your collections (say, from `Collection<T>` to one of your own collections which implements `Ilist<T>`). Notice that if you, for instance, apply `List<T>` operations, which are not prescribed by one of the interfaces, you need to declare your list of type `List<T>` for some type `T`.

Let us illustrate how this can be done in Program 45.23. The thing to notice is that the only place we refer to a list class (here `Collection<Animal>`()) is in line 9: new `Collection<Animal>`. All other places, as emphasized with **purple**, we use the interface `ICollection<Animal>`. If we, tomorrow, wish to change the representation of the animal collection, the only place to modify is line 9.

```
1  using System;
2  using System.Collections.Generic;
3  using System.Collections.ObjectModel;
4
5
6  class CollectionInterfaceDemo{
7
8    public static void Main(){
9      ICollection<Animal> lst = new Collection<Animal>();
10
11     // Add elements to the end of the empty list:
12     lst.Add(new Animal("Cat"));  lst.Add(new Animal("Dog", Sex.Female));
```

```
    lst.Add(new Animal("Mouse"));  lst.Add(new Animal("Rat"));
    lst.Add(new Animal("Mouse", Sex.Female));  lst.Add(new Animal("Rat"));
    lst.Add(new Animal("Herring", AnimalGroup.Fish, Sex.Female));
    lst.Add(new Animal("Eagle", AnimalGroup.Bird, Sex.Male));

    // Report in various ways on the animal collection:
    Print("Initial List", lst);
    ReportFemaleMale(lst);
    ReportGroup(lst);
  }

  public static void Print<T>(string explanation, ICollection<T> list){
    Console.WriteLine(explanation);
    foreach(T el in list)
      Console.WriteLine("{0, 3}", el);
    Console.WriteLine(); Console.WriteLine();
  }

  public static void ReportFemaleMale(ICollection<Animal> list){
    int numberOfMales = 0,
        numberOfFemales = 0;

    foreach(Animal a in list)
      if (a.Sex == Sex.Male) numberOfMales++;
      else if (a.Sex == Sex.Female) numberOfFemales++;

    Console.WriteLine("Males: {0}, Females: {1}",
                       numberOfMales, numberOfFemales);
  }

  public static void ReportGroup(ICollection<Animal> list){
    int numberOfMammals = 0,
        numberOfBirds = 0,
        numberOfFish = 0;

    foreach(Animal a in list)
      if (a.Group == AnimalGroup.Mammal) numberOfMammals++;
      else if (a.Group == AnimalGroup.Bird) numberOfBirds++;
      else if (a.Group == AnimalGroup.Fish) numberOfFish++;

    Console.WriteLine("Mammals: {0}, Birds: {1}, Fish: {2}",
                       numberOfMammals, numberOfBirds, numberOfFish);
  }

}
```

Program 45.23 *A program based on* ***`ICollection<Animal>`*** *- with a* ***`Collection<Animal>`****.*

On the accompanying slide we show versions of Program 45.23, which are tightly bound to the class `Collection<Animal>`, and we show a version in which we have replaced `Collection<Animal>` with `List<Animal>`.

45.19. References

[Big-O] Wikipedia: Big O Notation
http://en.wikipedia.org/wiki/Big_O_notation

46. Generic Dictionaries in C#

In the same style as our coverage of lists in Chapter 45 we will in this chapter discuss generic interfaces and classes for *dictionaries*. This covers the high-level concept of *associative arrays* and the low-level concept of *hash tables*.

46.1. Overview of Generic Dictionaries in C#

Lecture 12 - slide 24

A dictionary is a data structure that maps keys to values. A given key can have at most one value in the dictionary. In other words, the key of a key-value pair must be unique in the dictionary. A given value can be associated with many different keys.

At the conceptual level, a dictionary can be understood as an associative array (see Section 19.2) or as a collection of key-value pairs. In principle the collection classes from Chapter 45 can be used as an underlying representation. It is, however, convenient to provide a specialized interface to dictionaries which sets them apart from collections in general. In addition we often need good performance (fast lookup), and therefore it is more than justified to have special support for dictionaries in the C# libraries.

Figure 46.1 gives an overview of the generic interfaces and the generic classes of dictionaries. The figure is comparable with Figure 45.1 for collections. As such, the white boxes represent interfaces and the grey boxes represent classes. As it appears from Figure 46.1 we model dictionaries as `IEnumerables` (see Section 45.2) and `ICollections` (see Section 45.3) at the highest levels of abstractions. From the figure we can directly read that a dictionary ***is a*** `ICollection` of `KeyValuePairs`. (The **is a** relation is discussed in Section 25.2).

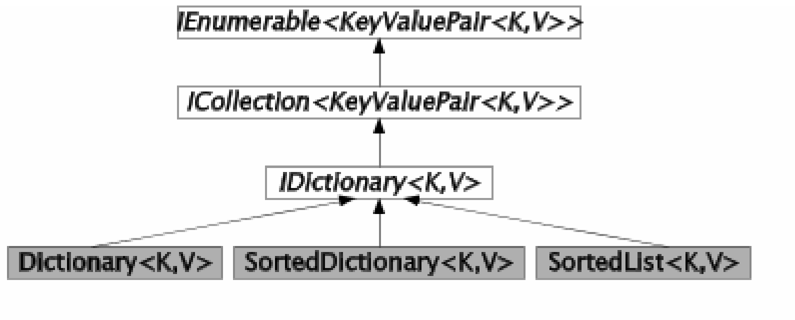

Figure 46.1 *The class and interface inheritance tree related to Dictionaries*

The symbol `K` stands for the type of keys, and the symbol `V` stands for the type of values. `KeyValuePair<K,V>` is a simple struct that aggregates a key and a value to a single object.

`Dictinonary<K,V>` is implemented in terms of a hashtable that maps objects of type `K` to objects of type `V`. `SortedDictinonary<K,V>` relies on binary search trees. `SortedList<K,V>` is based on a sorted arrays. More details can be found in Section 46.5. In Section 46.6 we review the time complexities of the operations of the three dictionary classes shown above.

46.2. The interface IDictionary<K,V>

Lecture 12 - slide 25

From Figure 46.1 we see that the interface `IDictionary<K,V>` is a subinterface of `ICollection<KeyValuePair<K,V>>`. We gave an overview of the generic interface `ICollection<T>` in Section 45.3. Because of this subinterface relationships we know that it is possible to use the operations `Contains`, `Add`, `Remove` on objects of type `KeyValuePair<K,V>`. Notice, however, that these operations are rather inconvenient because the generic class `KeyValuePair` is involved. Instead of `Add(new KeyValuePair(k,v))` we prefer another overload of `Add`, namely `Add(k,v)`. The mentioned operations `Contains`, `Add`, and `Remove` on `KeyValuePairs` are available in the `Dictionary` classes of Figure 46.1, but they are degraded to explicit interface implementations (see Section 31.8).

The following provides an overview of the operations in **`IDictionary<K,V>`**:

- *The operations prescribed in* **`ICollection<KeyValuePair<K,V>>`**
- *The operations prescribed in* **`IEnumerable<KeyValuePair<K,V>>`**
- V **`this`**[K key] - both getter and setter; the setter adds or mutates
- void **`Add`**(K key, V value) - only possible if key is not already present
- bool **`Remove`**(K key)
- bool **`ContainsKey`**(K key)
- bool **`TryGetValue`**(K key, out V value)
- `ICollection<K>`**`Keys`** - getter
- `ICollection<V>`**`Values`** - getter

`V this[K key]` is an indexer via which we can set and get a value of a given key by means of array notation (see Section 19.1). If `dict` is declared of type `IDictionary<K,V>` then the indexer notation allows us to express

```
valVar = dict[someKey];
dict[someKey] = someValue;
```

The first line accesses (gets/reads) the value associated with `someKey`. If no value is associated with `someKey` an `KeyNotFoundException` is thrown. The second line adds (sets/writes) an association between `someKey` and `someValue` to `dict`. If the association is already in the dictionary, the setter mutates the value associated with `someKey`.

The operation `Add(key,value)` adds an association between `key` and `value` to the dictionary. If the key is already associated with (another) value in the dictionary an `ArgumentException` will be thrown.

`Remove(key)` removes the association of `key` and its associated value. Via the value returned, the `Remove` operation signals if the removal was successful. `Remove` returns *false* if key is not present in the dictionary.

`ContainsKey(key)` tells if `key` is present in the dictionary.

The operation call `TryGetValue(key, valueVar)` accesses the value of `key`, and it passes the value via an output parameter (see Section 20.7). If no value is associated with key, the default value of type `V` (see Section 12.3) is passed back in the output parameter. This method is added of convenience. Alternatively, the indexer can be used in combination with `ContainsKey`.

The properties `Keys` and `Values` return collections of the keys and the values of a dictionary.

46.3. Overview of the class Dictionary<K,V>

Lecture 12 - slide 26

The generic class `Dictionary<K,V>` is based on hashtables. `Dictionary<K,V>` implements the interface `IDictionary<K,V>` as described in Section 46.2. Almost all methods and properties of `Dictionary<K,V>` are prescribed by the direct and indirect interfaces of the class. In the web version of the material we enumerate the most important operations of `Dictionary<K,V>`.

As it appears from the discussion of dictionaries above, it is necessary that two keys can be compared for equality. The equality comparison can be provided in several different ways. It is possible to pass an `EqualityComparer` object to the `Dictionary` constructor. Alternatively, we fall back on the *default equality comparer* of the key type *K*. The property `Comparer` of class `Dictionary<K,V>` returns the comparer used for key comparison in the current dictionary. See also the discussion of equality comparison in Section 42.9.

As already mentioned, a dictionary is implemented as a hash table. A hash table provides very fast access to the a value of a given key. Under normal circumstances - and with a good hash function - the run times of the access operations are constant (the run times do not depend on the size of the dictionary). Thus, the time complexity is O(1). Please consult Section 46.6 for more details on the efficiency of the dictionary operations.

46.4. Sample use of class Dictionary<K,V>

Lecture 12 - slide 27

In this section we will illustrate the use of dictionaries with a simple example. We go for a dictionary that maps objects of type `Person` to objects of type `BankAccount`. Given a `Person` object (the key) we wish to have efficient access to the person's `BankAccount` (the value).

The class `Person` is similar to Program 20.3. The class `BankAccount` is similar to Program 25.1. The exact versions of `Person` and `BankAccount`, as used in the dictionary example, can be accessed via the accompanying slide page, or via the program index of this lecture.

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

class DictionaryDemo{

  public static void Main(){

    IDictionary<Person, BankAccount> bankMap =
      new Dictionary<Person,BankAccount>(new PersonComparer());

    // Make bank accounts and person objects
    BankAccount ba1 =  new BankAccount("Kurt", 0.01),
                ba2 =  new BankAccount("Maria", 0.02),
                ba3 =  new BankAccount("Francoi", 0.03),
                ba4 =  new BankAccount("Unknown", 0.04);

    Person p1 = new Person("Kurt"),
```

```
            p2 = new Person("Maria"),
            p3 = new Person("Francoi");

    ba1.Deposit(100); ba2.Deposit(200); ba3.Deposit(300);

    // Populate the bankMap:
    bankMap.Add(p1, ba1);
    bankMap.Add(p2, ba2);
    bankMap.Add(p3, ba3);
    ReportDictionary("Initial map", bankMap);

    // Print Kurt's entry in the map:
    Console.WriteLine("{0}\n", bankMap[p1]);

    // Mutate Kurt's entry in the map
    bankMap[p1] = ba4;
    ReportDictionary("bankMap[p1] = ba4;", bankMap);

    // Mutate Maria's entry in the map. PersonComparer crucial!
    ba4.Deposit(400);
    bankMap[new Person("Maria")] = ba4;
    ReportDictionary("ba4.Deposit(400);  bankMap[new Person(\"Maria\")] = ba4;",
bankMap);

    // Add p3 yet another time to the map
    // Run-time error: An item with the same key has already been added.
/*  bankMap.Add(p3, ba1);
    ReportDictionary("bankMap.Add(p3, ba1);", bankMap);
 */

    // Try getting values of some given keys
    BankAccount ba1Res = null,
                ba2Res = null;
    bool res1 = false,
         res2 = false;
    res1 = bankMap.TryGetValue(p2, out ba1Res);
    res2 = bankMap.TryGetValue(new Person("Anders"), out ba2Res);
    Console.WriteLine("Account: {0}. Boolean result {1}", ba1Res, res1);
    Console.WriteLine("Account: {0}. Boolean result {1}", ba2Res, res2);
    Console.WriteLine();

    // Remove an entry from the map
    bankMap.Remove(p1);
    ReportDictionary("bankMap.Remove(p1);", bankMap);

    // Remove another entry - works because of PersonComparer
    bankMap.Remove(new Person("Francoi"));
    ReportDictionary("bankMap.Remove(new Person(\"Francoi\"));", bankMap);
  }

  public static void ReportDictionary<K, V>(string explanation,
                                            IDictionary<K,V> dict){
    Console.WriteLine(explanation);
    foreach(KeyValuePair<K,V> kvp in dict)
      Console.WriteLine("{0}: {1}", kvp.Key, kvp.Value);
    Console.WriteLine();
  }
}

public class PersonComparer: IEqualityComparer<Person>{

  public bool Equals(Person p1, Person p2){
    return (p1.Name == p2.Name);
  }

```

```
83    public int GetHashCode(Person p){
84      return p.Name.GetHashCode();
85    }
  }
```

Program 46.1 *A program working with* ***Dictionary<Person,BankAccount>***.

In line 8-9 we make the dictionary `bankMap` of type `Dictionary<Person,BankAccount>`. We pass an instance of class `PersonComparer`, see line 76-86, which implements `IEqualityComparer<Person>`. In line 11-19 we make sample `BankAccount` and `Person` objects, and in line 24-26 we populate the dictionary `bankMap`.

In line 30 we see how to access the bank account of person `p1` (Kurt). We use the provided indexer of the dictionary. In line 33 we mutate the `bankMap`: Kurt's bank account is changed from the one referenced by `ba1` to the one referenced by `ba4`. In line 38 we mutate Maria's bank account in a similar way. Notice, however, that that the relative weak equality of `Person` objects (implemented in class `PersonComparer`) implies that the new `person("Maria")` in line 38 is equal to the person referenced by `p2`, and therefore line 38 mutates the dictionary entry for Maria.

In line 43 we attempt add yet another entry for Francoi. This is illegal because there is already an entry for Francoi in the dictionary. If the comments around line 43 are removed, a run time error will occur.

In line 52-53 we illustrate `TryGetValue`. First, in line 52, we attempt to access Maria's account. The out parameter `baRes1` is assigned to Maria's account and `true` is returned from the method. In line 53 we attempt to access the account of a brand new `Person` object, which has no bank account in the dictionary. `null` is returned through `ba2Res`, and `false` is returned from the method.

Finally, in line 58-64 we remove entries from the dictionary by use of the `Remove` method. First Kurt's entry is removed after which Francoi's entry is removed.

The output of the program is shown in Listing 46.2 (only on web).

Exercise 12.3. *Switching from Dictionary to SortedDictionary*

The program on this slide instantiates a **`Dictionary<Person,BankAccount>`**. As recommended earlier in this lecture, we should work with the dictionary via a variable of the interface type **`IDictionary<K,V>`**.

You are now asked to replace **`Dictionary<Person,BankAccount>`** with **`SortedDictionary<Person,BankAccount>`** in the above mentioned program.

This causes a minor problem. Identify the problem, and fix it.

Can you tell the difference between the output of the program on this slide and the output of your revised program?

You can access the `BankAccount` and `Person` classes in the web version of the material.

46.5. Notes about Dictionary Classes

Lecture 12 - slide 28

As can be seen from Figure 46.1 several different generic classes implement the `IDictionary<K,V>` interface. `Dictionary<K,V>`, as discussed in Section 46.3 and Section 46.4 is based on a hash table representation. `SortedDictionary<K,V>` is based on a binary tree, and (as the name signals) `SortedList<K,V>` is based on an array of key/value pairs, sorted by keys.

The following provides an itemized overview of the three generic dictionary classes.

- Class **`Dictionary<K,V>`**
    - Based on a hash table
    - Requires that the keys in type **`K`** can be compared by an **`Equals`** operation
    - Key values should not be mutated
    - The efficiency of class dictionary relies on a good hash function for the key type **`K`**
        - Consider overriding the method **`GetHashCode`** in class `K`
    - A dictionary is enumerated in terms of the struct **`KeyValuePair<K,V>`**
- Class **`SortedDictionary<K,V>`**
    - Based on a binary search tree
    - Requires an **`IComparer`** for keys of type **`K`** - for ordering purposes
        - Provided when a sorted dictionary is constructed
- Class **`SortedList<K,V>`**
    - Based on a sorted collection of key/value pairs
        - A resizeable array
    - Requires an **`IComparer`** for keys, just like **`SortedDictionary<K,V>`**.
    - Requires less memory than **`SortedDictionary<K,V>`**.

When you have to chose between the three dictionary classes the most important concern is the different run time characteristics of the operations of the classes. The next section provides an overview of these.

46.6. Time complexity overview: Dictionary classes

Lecture 12 - slide 29

We will now review the time complexities of the most important dictionary operations. This is done in the same way as we did for collections (lists) in Section 45.17. We will assume that we work on a dictionary that holds *n* entries of key/value pairs.

Operation	**`Dictionary<K,V>`**	**`SortedDictionary<K,V>`**	**`SortedList<K,V>`**
`this[key]`	*O(1)*	*O(log n)*	*O(log n) or O(n)*
`Add(key,value)`	*O(1) or O(n)*	*O(log n)*	*O(n)*
`Remove(key)`	*O(1)*	*O(log n)*	*O(n)*
`ContainsKey(key)`	*O(1)*	*O(log n)*	*O(log n)*
`ContainsValue(value)`	*O(n)*	*O(n)*	*O(n)*

Table 46.1 *Time complexities of important operations in the classes* `Dictionary<K,V>`, `SortedDictionary<K,V>`, *and* `SortedList<K,V>`.

As noticed in Section 46.5 an object of type `Dictionary<K,V>` is based on hash tables. Eventually, it will be necessary to enlarge the hashtable to hold new elements. It is good wisdom to enlarge the hashtable when it becomes half full. The *O(1) or O(n)* time complexity for `Add` reflects that a work proportional to *n* is needed when it becomes necessary to enlarge the hash table.

Most operations on the binary tree representation of `SortedDictionary<K,V>` are logarithmic in *n*. The only exception (among the operations listed in the table) is `ContainsValue`, which in the worst case requires a full tree traversal.

In `SortedList<K,V>` the indexer is efficient, *O(log n)* when an existing item is mutated. If use of the indexer causes addition of a new entry, the run time is the same as the run time of `Add`. Adding elements to a sorted list requires, in average, that half of the elements are pushed towards the end of the list in order to create free space for the new entry. This is an *O(n)* operation. `Remove` is symmetric, pulling elements towards the beginning of the list, and therefore also *O(n)*. `ContainsKey` is efficient because we can do binary search on the sorted list. `ContainsValue` requires linear search, and therefore it is an *O(n)* operation.

Given the table in Table 46.1 it is tempting to conclude that `Dictionary<K,V>` is the best of the three classes. Notice, however, that the difference between a constant run time *c1* and *c2 log(n)* is not necessarily significant. If the constant *c1* is large and the constant *c2* is small, the binary tree may be an attractive alternative. Furthermore, we know that the hashtable will be slow when it is almost full. In that case more and more collisions can be expected. At some point in time the hash table will stop working if it is not resized. This is not an issue if we work with balanced binary trees. Finally, the hashtable depends critically on a good hash function, preferable programmed specifically for the key type `K`. This is not an issue if we use binary trees.

47. Non-generic Collections in C#

This is a short chapter in which we discuss the non-generic collection classes. You may encounter use of these classes in many older C# programs. In Section 44.1 these collection classes were called *first generation collection classes*.

47.1. The non-generic collection library in C#

Lecture 12 - slide 31

The overview of the non-generic collection interfaces and classes in Figure 47.1 is a counterpart to the sum of Figure 45.1 and Figure 46.1. The white boxes represent interfaces and the grey boxes represent classes. Most classes and interfaces shown in Figure 46.1 belong to the namespace `System.Collections`.

The non-generic collection classes store data of type **`Object`**

As the most important characteristics, the elements of the lists are of type `Object`. Both keys and values of dictionaries are `Objects`. Without use of type parametrization, there are no means to constraint the data in collections to of a more specific type. Thus, if we for instance work with a collection of bank accounts, we cannot statically guarantee that all elements of the collection are bank accounts. We may accidentally insert an object of another type. We will find the error at runtime. Most likely, an exception will be raised when we try to cast an `Object` to `BankAccount`.

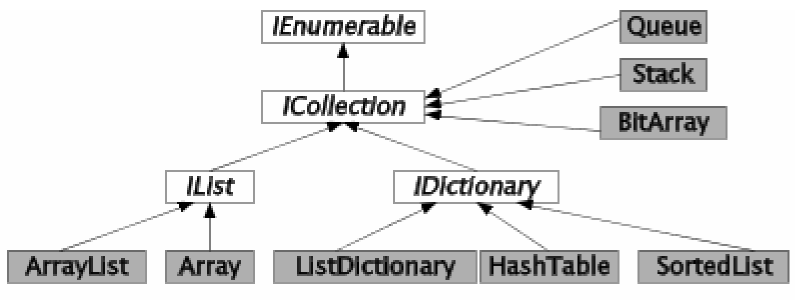

Figure 47.1 *The class and interface inheritance tree related to collections*

The `IEnumerable`, `ICollection`, `IList` and `IDictionary` interfaces of Figure 47.1 are natural counterparts to the generic interfaces `IEnumerable<T>`, `ICollection<T>`, `IList<T>` and `IDictionary<K,V>`.

The class `ArrayList` corresponds to `List<T>`. As such, `ArrayList` is a class with a rich repertoire of operations for searching, sorting, and range operations. `ArrayList` is undoubtedly the most widely used collection class in C# 1.0 programs.

The `Array` class shown next to `ArrayList` in Figure 47.1 deserves some special clarification. It belongs to the `System` namespace. You cannot instantiate class `Array` in your programs, because `Array` is an abstract class. And you cannot use `Array` as a superclass of one of your own classes. So, class `Array` seems pretty useless. At least it is fair to state the class `Array` is rather special compared to the other classes in Figure 47.1.

Let us now explain the role of class `Array`. As mentioned earlier, see Section 28.2 , class `Array` acts as the superclass of all "native" array types in C#. (See the discussion of arrays in Section 6.4). Consequently, all the nice operation in `System.Array` can be used on all "native" arrays that you use in your C# programs. If, for instance, we have the array declarations

```
int[] ia = new int[3];
string[] sa = new string[5,6];
BankAccount[] baa = new BankAccount[10];
```

the following are legal expressions

```
ia.Length
a.Rank
Array.BinarySearch(ia, 5)
Array.Find(sa, IsPalindrome)
Array.Sort(baa)
```

In the `Array` class, you should pay attention to the (overloaded) static method `CreateInstance`, which allows for programmatic creation on an arbitrary array. The `Array` instance methods `GetValue` and `SetValue` allow us to access elements in arbitrary arrays - independent of element type and rank.

When we talk about "native arrays" in C# we refer to the array concept implemented in the language as such. The compiler provides special support for these native arrays. In contrast, generic and non-generic collections are provided via the class library. The C# compiler and the C# interpreter do not have particular knowledge or support of the collection classes. We could have written these classes ourselves! It is interesting to notice that the native arrays, as derived from class `Array` in Figure 47.1, are type safe. The type safeness of native arrays is due to the special support by the compiler, which allows for declaration of the element types of the arrays (see the examples of `int`, `string`, and `BankAccount` arrays above).

The class `HashTable` in Figure 47.1 corresponds to the generic class `Dictionary<K,V>`, see Section 46.3 and Section 46.4).

The class `ListDictionary`, which belongs to the namespace `System.Collections.Specialized`, has no natural generic counterpart. `ListDictionary` is based on linear search in an unordered collection of key/value pairs. `ListDictionary` should therefore only be used for small dictionaries.

As the name suggests, class `SortedList` corresponds to `SortedList<K,V>`. Both rely on a (linear) list representation, sorted by keys.

The class `BitArray` is - by nature - a non-generic collection class. The binary digit 1 is represented as boolean *true*, and the binary digit 0 is represented as boolean *false*. `BitArray` provides a compact representation of a bit arrays. In the context of indexers, see Program 19.4, we have earlier discussed a partial reproduction of the class `BitArray`.

In addition to the types shown in Figure 47.1 there exist some specialized collections in the namespace `System.Collections.Specialized`. As an example, the class `StringCollection` is a collection of strings. The class `CollectionBase` in the namespace `System.Collection` is intended as the superclass of new, specialized collection classes. In the documentation of this class, an example shows how to define an `Int16Collection` as a subclass of `CollectionBase`. Needless to say, *all these classes are obsolete* relative to both C#2.0 and C#3.0. As of today, the classes may be necessary for backward compatibility, but, unfortunately, they also add to the complexity of the .NET class libraries.

48. Patterns and Techniques

In earlier parts of this material (Section 31.6 and Section 45.2) we have at length discussed enumerators in C#, including their relationship to **foreach** loops.

In this section we first briefly rephrase this to the design pattern known as ***Iterator***. Following that we will show how to implement iterators (enumerators) with use of **yield return**, which is a variant of the **return** statement.

48.1. The Iterator Design Pattern

Lecture 12 - slide 34

The ***Iterator*** design pattern provides sequential access to an aggregated collection. At an overall level, an iterator

- Provides for a smaller interface of the collection class
  - All members associated with traversals have been refactored to the iterator class
- Makes it possible to have several simultaneous traversals
- Does not reveal the internal representation of the collection

As we have seen in Section 31.6 and Section 45.2, traversal of a collection requires a few related operations, such as `Current`, `MoveNext`, and `Reset`. We could imagine a slightly more advanced iterator which could move backwards as well. With use of iterators we have factored these operations out of the collection classes, and organized them in iterators (enumerators). With this refactoring, a collection can be asked to deliver an iterator:

```
aCollection.GetEnumerator()
```

Each iterator maintains the state, which is necessary to carry out a traversal of a collection. If we need two independent, simultaneous traversals we can ask for two iterators of the collections. This could, for instance be used to manage simultaneous iteration from both ends of a list.

In more primitive collections, such as linked lists (see Section 45.14) it is necessary to reveal the object structure that keeps the list together. (In `LinkedList<T>` this relates to the details of `LinkedListNode<T>` instances). With use of iterators it is not necessary to reveal such details. An iterator is an encapsulated, abstract representation of some state that manages a traversal. The concrete representation of this state is not leaked to clients. This is very satisfactory in an object-oriented programming context.

Iterators (enumerators) are typically used via foreach loops. As an alternative, it is of course also possible to use the operations in the `IEnumerator` interface directly to carry out traversals. Exercise 12.4 is a opportunity to train such a more direct use of iterators.

Exercise 12.4. *Explicit use of iterator - instead of using foreach*

In this program we will make direct use of an iterator (an enumerator) instead of traversing with use of foreach.

In the animal collection program, which we have seen earlier in this lecture, we traverse the animal collections several times with use of foreach. Replace each use of foreach with an application of an iterator.

48.2. Making iterators with yield return

Lecture 12 - slide 35

In this section we will show how to use the special-purpose **yield return** statement to define iterators, or as they are called in C#, enumerators. First, we will program a very simple collection of up to three, fixed values. Next we will revisit the integer sequence enumeration, which can be found in Section 58.3.

In Program 48.1 we will program a collection class, called `GivenCollection`, which just covers zero, one, two or three values of some arbitrary type `T`. As a simpleminded approach, we represent these `T` values with three instance variables of type `T`, and with three boolean variables which tells if the corresponding `T` values are present. As an invariant, the instance variables are filled from the lower end. It would be tempting to use the type `T?` instead of `T`, and the value `null` for a missing value. But this is not possible if `T` is class.

It is important that the class `GivenCollection` implements the generic interface **`IEnumerable<T>`**. Because this interface, in turn, implements the non-generic `IEnumerable`, we must both define the generic and the non-generic `GetEnumerator` method. The latter must be defined as an explicit interface (see Section 31.8), in order not to conflict with the former. If we forget the non-generic `GetEnumerator`, we get a slightly misleading error message:

> '**`GivenCollection<T>`**' does not implement interface member 'System.Collections.IEnumerable.GetEnumerator()'.
> '**`GivenCollection<T>`**' is either static, not public, or has the wrong return type.

This message can cause a lot of headache, because the real problem (the missing, non-generic `GetEnumerator` method) is slightly camouflaged in the error message.

The implementation of the non-generic enumerator just delegates its work to the generic version.

The implementation of the generic `Enumerator` method uses the **yield return** statement. Let us assume that an instance of `GivenCollection<T>` holds three `T` values (in `first`, `second`, and `third`). The three boolean variables `firstDefined`, `secondDefined`, and `thirdDefined` are all true. The `GetEnumerator` method has three yield return statements in sequence (see line 50-52). By means of these, `GetEnumerator` can return three values before it is done. This is entirely different from a normal method, which only returns once (after which it is done). The `GetEnumerator` in class `GivenCollection` acts as a coroutine in relation to its calling place (which is the **foreach** statement in the client program Program 48.2). A coroutine can resume execution at the place where execution stopped in an earlier call. A normal method always (re)starts from its first statement each time it is called.

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

public class GivenCollection<T> : IEnumerable<T>{

  private T first, second, third;
  private bool firstDefined, secondDefined, thirdDefined;

  public GivenCollection(){
    this.firstDefined = false;
    this.secondDefined = false;
    this.thirdDefined = false;
  }

  public GivenCollection(T first){
    this.first = first;
    this.firstDefined = true;
    this.secondDefined = false;
    this.thirdDefined = false;
  }

  public GivenCollection(T first, T second){
    this.first = first;
    this.second = second;
    this.firstDefined = true;
    this.secondDefined = true;
    this.thirdDefined = false;
  }

  public GivenCollection(T first, T second, T third){
    this.first = first;
    this.second = second;
    this.third = third;
    this.firstDefined = true;
    this.secondDefined = true;
    this.thirdDefined = true;
  }

  public int Count(){
    int res;
    if (!firstDefined) res = 0;
    else if (!secondDefined) res = 1;
    else if (!thirdDefined) res = 2;
    else res = 3;
    return res;
  }

  public IEnumerator<T> GetEnumerator(){
    if (firstDefined) yield return first;
    if (secondDefined) yield return second;  // not else
    if (thirdDefined) yield return third;    // not else
  }

  IEnumerator IEnumerable.GetEnumerator(){
    return GetEnumerator();
  }

}
```

Program 48.1 *A collection of up to three instance variables of type T - with an iterator.*

In Program 48.2 we show a simple program that instantiates a `GivenCollection` of the integers 7, 5, and 3. The **foreach** loop in line 11-12 traverses the three corresponding instance variables, and prints each of them.

```
1  using System;
2
3  class Client{
4
5    public static void Main(){
6
7       GivenCollection<int> gc = new GivenCollection<int>(7,5,3);
8
9       Console.WriteLine("Number of elements in givenCollection: {0}",
10                         gc.Count());
11      foreach(int i in gc){      // Output:  7 5 3
12        Console.WriteLine(i);
13      }
14
15   }
16
17 }
```

Program 48.2 *A sample iteration of the three instance variable collection.*

Exercise 12.5. *The iterator behind a yield*

Reprogram the iterator in class `GivenCollection` without using the **yield return** statement in the `GetEnumerator` method.

Let us now revisit the integer enumeration classes of Section 58.3. The main point in our first discussion of these classes was the ***Composite*** design pattern, cf. Section 32.1, as illustrated in Figure 58.1 of Section 58.3. The three classes `IntInterval`, `IntSingular`, and `IntCompSeq` all inherit the abstract class `IntSequece`. You can examine the abstract class `IntSequence` in Program 58.9 in the appendix of this material. The three concrete subclasses were programmed in Program 58.10, Program 58.11, and Program 58.12.

The `GetEnumerator` methods of `IntInterval`, `IntSingular`, and `IntCompSeq` are all emphasized below in Program 48.3, Program 48.4, and Program 48.5. Notice the use of **yield return** in all of them.

In Program 48.3 the if-else of `GetEnumerator` in line 19-24 distinguishes between increasing and decreasing intervals. The `GetEnumerator` method of `IntSingular` is trivial. The `GetEnumerator` method of `IntCompSeq` in Program 48.5 is surprisingly simple - at least compared with the counterpart in Program 58.12. The two foreach statements (in sequence) in line 19-22 activate all the machinery, which we programmed manually in Program 58.12. This includes recursive access to enumerators of composite sequences.

The simplicity of enumerators, programmed with yield return, is noteworthy compared to all the underlying stuff of explicitly programmed classes that implement the interface `IEnumerator`.

Iterators (iterator blocks), programmed with **`yield return`**, are only allowed to appear in methods that implement an enumerator or an enumerable interface (such as `IEnumerator` or `IEnumerator` and their generic counterparts). Such methods are handled in a very special way by the compiler, and a number of restrictions apply to these methods. The compiler generates all the machinery, which we program ourselves when a class implements the enumerator or enumerable interfaces. Methods with iterator blocks that implement and enumerator or an enumerable interface return an enumerator object, on which the `MoveNext` can be called a number of times. For more details on iterators please consult Section 10.14 in the C# 3.0 Language Specification [csharp-3-spec].

```
public class IntInterval: IntSequence{

  private int from, to;

  public IntInterval(int from, int to){
    this.from = from;
    this.to = to;
  }

  public override int? Min{
    get {return Math.Min(from,to);}
  }

  public override int? Max{
    get {return Math.Max(from,to);}
  }

  public override IEnumerator GetEnumerator (){
    if (from < to)
     for(int i = from; i <= to; i++)
       yield return i;
    else
     for(int i = from; i >= to; i--)
       yield return i;
  }

}
```

Program 48.3 *The class IntInterval - Revisited.*

```
public class IntSingular: IntSequence{

  private int it;

  public IntSingular(int it){
    this.it = it;
  }

  public override int? Min{
    get {return it;}
  }

  public override int? Max{
    get {return it;}
  }

  public override IEnumerator GetEnumerator(){
    yield return it;
  }
}
```

Program 48.4 *The class IntSingular - Revisited.*

```
public class IntCompSeq: IntSequence{

  private IntSequence s1, s2;

  public IntCompSeq(IntSequence s1, IntSequence s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  public override int? Min{
    get {return (s1.Min < s2.Min) ? s1.Min : s2.Min;}
  }

  public override int? Max{
    get {return (s1.Max > s2.Max) ? s1.Max : s2.Max;}
  }

  public override IEnumerator GetEnumerator (){
    foreach(int i in s1)
      yield return i;
    foreach(int i in s2)
      yield return i;
  }

}
```

Program 48.5 *The class IntCompSeq - Revisited.*

In the web edition of the material we show a sample client program that contains a couple of `IntSequences`.

48.3. References

[Csharp-3-spec] "The C# Language Specification 3.0",