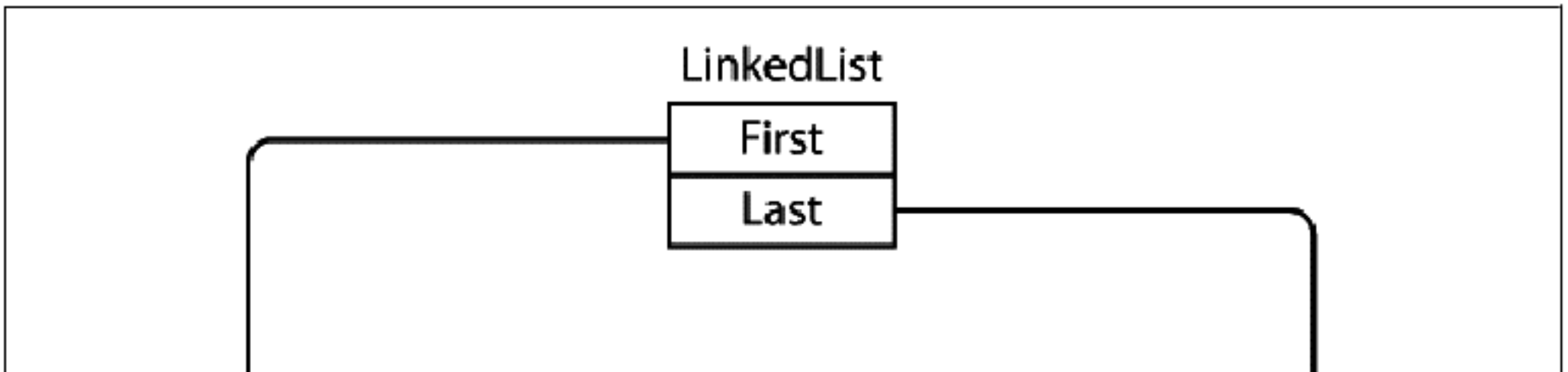
```
public void AddFirst(LinkedListNode<T> node);
public LinkedListNode<T> AddFirst (T value);
```

```
public void AddLast (LinkedListNode<T> node);
public LinkedListNode<T> AddLast (T value);
```



```
public void AddAfter (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddAfter (LinkedListNode<T> node, T value);

public void AddBefore (LinkedListNode<T> node, LinkedListNode<T> newNode);
public LinkedListNode<T> AddBefore (LinkedListNode<T> node, T value);
```



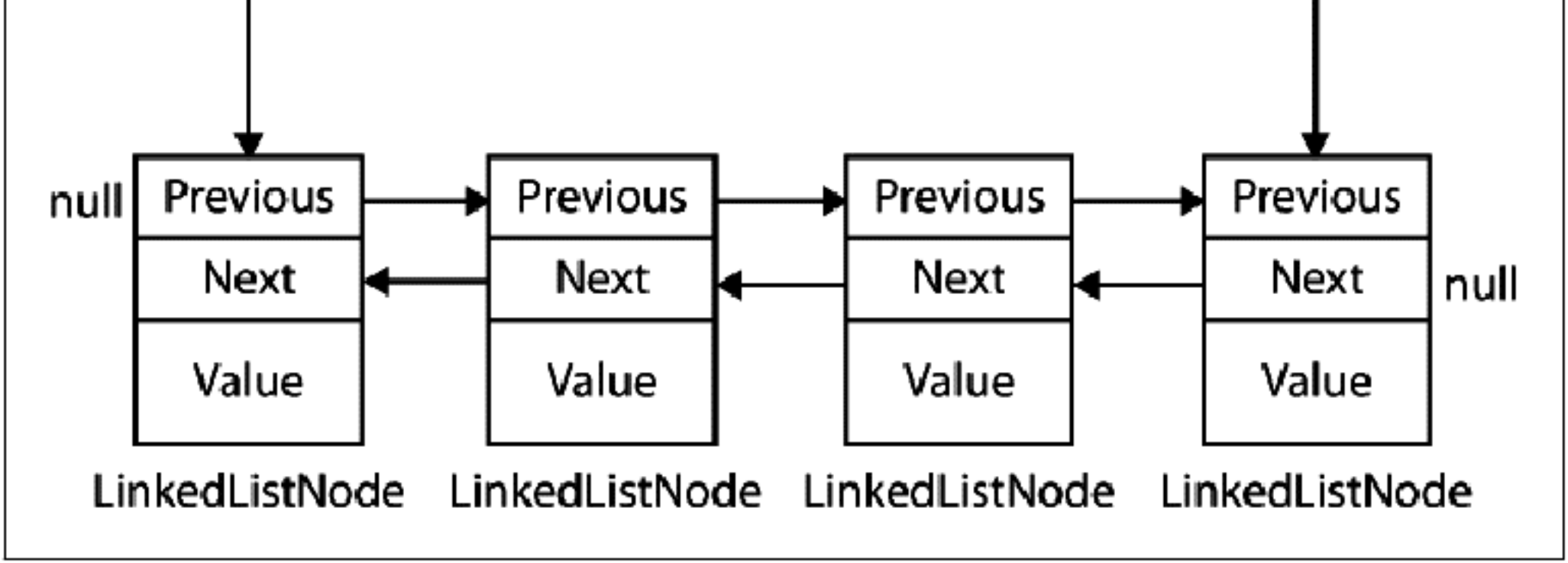


Figure 7-4. LinkedList<T>

Similar methods are provided to remove elements:

```
public void Clear();

public void RemoveFirst();
public void RemoveLast();

public bool Remove (T value);
public void Remove (LinkedListNode<T> node);
```

```
public void Remove (LinkedListNode<T> node);
```

`LinkedList<T>` has internal fields to keep track of the number of elements in the list, as well as the head and tail of the list. These are exposed in the following public properties:

```
public int Count { get; }                          // Fast
public LinkedListNode<T> First { get; }            // Fast
public LinkedListNode<T> Last { get; }             // Fast
```

`LinkedList<T>` also supports the following searching methods (each requiring that the list be internally enumerated):

```
public bool Contains (T value);
public LinkedListNode<T> Find (T value);
public LinkedListNode<T> FindLast (T value);
```

Finally, `LinkedList<T>` supports copying to an array for indexed processing and obtaining an enumerator to support the **foreach** statement:

```
public void CopyTo (T[] array, int index);
public Enumerator<T> GetEnumerator();
```

Here's a demonstration on the use of LinkedList<string>:

```
var tune = new LinkedList<string>();
tune.AddFirst ("do");                                   // do
tune.AddLast ("so");                                    // do - so

tune.AddAfter (tune.First, "re");
tune.AddAfter (tune.First.Next, "mi");
tune.AddBefore (tune.Last, "fa");

tune.RemoveFirst();
tune.RemoveLast();

// do - re - so
// do - re - mi - so
```

```
// do - re - mi - so
// do - re - mi - fa - so
// re - mi - fa - so
// re - mi - fa
```

```
LinkedListNode<string> miNode = tune.Find ("mi");
tune.Remove (miNode);                                   // re - fa
tune.AddFirst (miNode);                                 // mi - re - fa
```

```
foreach (string s in tune) Console.WriteLine (s);
```

Queue<T> and Queue

Queue<T> and Queue are first-in first-out (FIFO) data structures, providing methods to Enqueue (add an item to the tail of the queue) and Dequeue (retrieve and remove the item at the head of the queue). A Peek method is also provided to return the element at the head of the queue without removing it, and a Count property (useful

element at the head of the queue without removing it, and a `Count` property (useful in checking that elements are present before dequeuing).

Although queues are enumerable, they do not implement `IList<T>`/`IList`, since members cannot be accessed directly by index. A `ToArray` method is provided, however, for copying the elements to an array where they can be randomly accessed:

```
public class Queue<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Queue();
  public Queue (IEnumerable<T> collection);   // Copies existing elements
  public Queue (int capacity);                // To lessen auto-resizing
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public T Dequeue();
  public void Enqueue (T item);
  public Enumerator<T> GetEnumerator();       // To support foreach
  public T Peek();
  public T[] ToArray();
  public void TrimExcess();
}
```

```
  public void TrimExcess();
}
```


The following is an example of using `Queue<int>`:

```
var q = new Queue<int>();
q.Enqueue (10);
q.Enqueue (20);
int[] data = q.ToArray();          // Exports to an array
```

```
q.Enqueue (20);
int[] data = q.ToArray();          // Exports to an array
Console.WriteLine (q.Count);       // "2"
Console.WriteLine (q.Peek());      // "10"
```



```
Console.WriteLine (q.Dequeue());
Console.WriteLine (q.Dequeue());
Console.WriteLine (q.Dequeue());

// "10"
// "20"
// throws an exception (queue empty)
```

Queues are implemented internally using an array that's resized as required—much like the generic `List` class. The queue maintains indexes that point directly to the

Queues are implemented internally using an array that's resized as required—much like the generic `List` class. The queue maintains indexes that point directly to the head and tail elements; therefore, enqueuing and dequeuing are extremely quick operations (except when an internal resize is required).

Stack<T> and Stack

`Stack<T>` and `Stack` are last-in first-out (LIFO) data structures, providing methods to `Push` (add an item to the top of the stack) and `Pop` (retrieve and remove an element from the top of the stack). A nondestructive `Peek` method is also provided, as is a `Count` property and a `ToArray` method for exporting the data for random access:

```
public class Stack<T> : IEnumerable<T>, ICollection, IEnumerable
{
  public Stack();
  public Stack (IEnumerable<T> collection);   // Copies existing elements
  public Stack (int capacity);                // Lessens auto-resizing
  public void Clear();
  public bool Contains (T item);
  public void CopyTo (T[] array, int arrayIndex);
  public int Count { get; }
  public Enumerator<T> GetEnumerator();       // To support foreach
```

```
  public int Count { get; }
  public Enumerator<T> GetEnumerator();       // To support foreach
  public T Peek();
  public T Pop();
  public void Push (T item);
  public T[] ToArray();
  public void TrimExcess();
}
```

The following example demonstrates `Stack<int>`:

```
var s = new Stack<int>();
s.Push (1);                          //              Stack = 1
s.Push (2);                          //              Stack = 1,2
s.Push (3);                          //              Stack = 1,2,3
Console.WriteLine (s.Count);         // Prints 3
Console.WriteLine (s.Peek());        // Prints 3,  Stack = 1,2,3
Console.WriteLine (s.Pop());         // Prints 3,  Stack = 1,2
Console.WriteLine (s.Pop());         // Prints 2,  Stack = 1
Console.WriteLine (s.Pop());         // Prints 1,  Stack = <empty>
Console.WriteLine (s.Pop());         // throws exception
```

Stacks are implemented internally with an array that's resized as required, as with

Stacks are implemented internally with an array that's resized as required, as with `Queue<T>` and `List<T>`.

BitArray

A `BitArray` is a dynamically sized collection of compacted `bool` values. It is more memory-efficient than both a simple array of `bool` and a generic `List` of `bool`, because it uses only one bit for each value, whereas the `bool` type otherwise occupies one byte for each value:


```
public sealed class BitArray : ICollection, IEnumerable, ICloneable
{
  // Constructors
  public BitArray (BitArray bits);    // An existing BitArray to copy
  public BitArray (int length);       // Capacity, in bits
  public BitArray (bool[] values);
  public BitArray (byte[] bytes);
  public BitArray (int[] values);
```

```
public BitArray (int[] values);
public BitArray (int length, bool defaultValue);
```

```
// To get/set value
public bool this [int index] { get; set; }
public bool Get  (int index);
public void Set  (int index, bool value);
public void SetAll (bool value);

// Bitwise operators
public BitArray Not();
public BitArray And (BitArray value);
public BitArray Or  (BitArray value);
public BitArray Xor (BitArray value);

// Copying
public void CopyTo (Array array, int index);
```

```
public void CopyTo (Array array, int index);
public object Clone();

  // Other
  public IEnumerator GetEnumerator();
  public int  Count                { get; }
  public int  Length               { get; set; }
  public bool IsReadOnly           { get; }
  public bool IsSynchronized { get; }
  public object SyncRoot           { get; }
}
```

The following is an example of using the `BitArray` class:

```
var bits = new BitArray(2);
bits[1] = true;
bits.Xor (bits);                    // Bitwise exclusive-OR bits with itself
```

```
bits[1] = true;
bits.Xor (bits);                // Bitwise exclusive-OR bits with itself
Console.WriteLine (bits[1]);    // False
```

HashSet<T> and SortedSet<T>

`HashSet<T>` and `SortedSet<T>` are generic collections new to Framework 3.5 and 4.0, respectively. Both have the following distinguishing features:

-
-
-

Their `Contains` methods execute quickly using a hash-based lookup.

They do not store duplicate elements and silently ignore requests to add duplicates.

You cannot access an element by position.

`SortedSet<T>` keeps elements in order, whereas `HashSet<T>` does not.

The commonality of these types is captured by the interface `ISet<T>`.

For historical reasons, `HashSet<T>` lives in *System.Core.dll* (whereas `SortedSet<T>` and `ISet<T>` live in *System.dll*).

(whereas SortedSet<T> and ISet<T> live in *System.dll*).

HashSet<T> is implemented with a hashtable that stores just keys; SortedSet<T> is implemented with a red/black tree.

Here's the definition for HashSet<T>:

```
public class HashSet<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  // Constructors
  public HashSet();
  public HashSet (IEnumerable<T> collection);
  public HashSet (IEqualityComparer<T> comparer);
  public HashSet (IEnumerable<T> collection, IEqualityComparer<T> comparer);
```

```
// Testing for membership
public bool Contains (T item);

// Adding / removing
public bool Add    (T item);
public bool Remove (T item);
public int RemoveWhere (Predicate<T> match);
```

```
public int RemoveWhere (Predicate<T> match);
public void Clear();

// Set operations - destructive
public void UnionWith          (IEnumerable<T> other);
public void IntersectWith      (IEnumerable<T> other);
public void ExceptWith         (IEnumerable<T> other);
public void SymmetricExceptWith (IEnumerable<T> other);

// Set operations - bool
public bool IsSubsetOf         (IEnumerable<T> other);
public bool IsProperSubsetOf   (IEnumerable<T> other);
public bool IsSupersetOf       (IEnumerable<T> other);
public bool IsProperSupersetOf (IEnumerable<T> other);
public bool Overlaps           (IEnumerable<T> other);
public bool SetEquals          (IEnumerable<T> other);
```

```
// Adds
```

```
  // Adds
  // Removes
  // Removes
  // Removes

  // Other
  public int Count { get; }
  public IEqualityComparer<T> Comparer { get; }
  public void CopyTo (T[] array);
  public void CopyTo (T[] array, int arrayIndex);
  public void CopyTo (T[] array, int arrayIndex, int count);
  public void TrimExcess();
  public static IEqualityComparer<HashSet<T>> CreateSetComparer();
}
```

`SortedSet<T>` offers the same set of members, plus the following:

```
public virtual SortedSet<T> GetViewBetween (T lowerValue, T upperValue)
public IEnumerable<T> Reverse()
```

```
public T Min { get; }
public T Max { get; }
```

`SortedSet<T>` also accepts an optional `IComparer<T>` in its constructor (rather than an *equality comparer*).

The following constructs a `HashSet<char>` from an existing collection, tests for membership, and then enumerates the collection (notice the absence of duplicates):

```
var letters = new HashSet<char> ("the quick brown fox");

Console.WriteLine (letters.Contains ('t'));
Console.WriteLine (letters.Contains ('j'));

foreach (char c in letters) Console.Write (c);
```

```
// true
// false
// the quickbrownfx
```

(The reason we can pass a `string` into `HashSet<char>`'s constructor is because `string` implements `IEnumerable<char>`.)

Here's an example of loading the same letters into a `SortedSet<char>`:

```
var letters = new SortedSet<char> ("the quick brown fox");
foreach (char c in letters) Console.Write (c);   //  bcefhiknoqrtuwx
```

Following on from this, we can obtain the letters between *f* and *j* as follows:

```
foreach (char c in letters.GetViewBetween ('f', 'j'))
  Console.Write (c);                                    //  fhk
```

The destructive set operators modify the original collection. `UnionWith` adds all the elements in the second set to the original set (excluding duplicates). `Intersects`

The destructive set operators modify the original collection. `UnionWith` adds all the elements in the second set to the original set (excluding duplicates). `IntersectsWith` removes the elements that are not in both sets. We remove all the vowels from our set of characters in the following code:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.IntersectWith ("aeiou");
foreach (char c in letters) Console.Write (c);     // euio
```

`ExceptWith` removes the specified elements from the source set. Here, we strip all vowels from the set:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.ExceptWith ("aeiou");
foreach (char c in letters) Console.Write (c);     // th qckbrwnfx
```

`SymmetricExceptWith` removes all but the elements that are unique to one set or the other:

```
var letters = new HashSet<char> ("the quick brown fox");
letters.SymmetricExceptWith ("the lazy brown fox");
foreach (char c in letters) Console.Write (c);     // quicklazy
```

Because `HashSet<T>` and `SortedSet<T>` implement `IEnumerable<T>`, you can use another set as the argument to any of the set operation methods.

Dictionaries

Dictionaries

A dictionary is a collection in which each element is a key/value pair. Dictionaries are most commonly used for lookups and sorted lists.

The Framework defines a standard protocol for dictionaries, via the interfaces `IDictionary` and `IDictionary <TKey, TValue>`, as well as a set of general-purpose dictionary classes. The classes each differ in the following regard:

-
-
-
-

Whether or not items are stored in sorted sequence

Whether or not items can be accessed by position (index) as well as by key

Whether generic or nongeneric

Their performance when large

Table 7-1 summarizes each of the dictionary classes and how they differ in these

Table 7-1 summarizes each of the dictionary classes and how they differ in these respects. The performance times are in milliseconds, to perform 50,000 operations on a dictionary with integer keys and values, on a 1.5 GHz PC. (The differences in performance between generic and nongeneric counterparts using the same underlying collection structure are due to boxing, and show up only with value-type elements.)

Table 7-1. Dictionary classes

Type	Internal structure	Retrieve by index?	Memory overhead (avg. bytes per item)	Speed: random insertion	Speed: sequential insertion	Speed: retrieval by key
Unsorted						
`Dictionary <K,V>`						
`Hashtable`						
`ListDictionary`						
`OrderedDictionary`						

`OrderedDictionary`	
Sorted	
Hashtable	
Hashtable Linked list Hashtable + array	
No	
No No Yes	

Yes

22
38
36
59

30
50

30
50

20
30

30

50,000

70

50,000

70

50,000

40

`SortedDictionary <K,V>`	Red/black tree	No	20	130	100	120
`SortedList <K,V>`	2xArray	Yes	2	3,300	30	40
`SortedList`	2xArray	Yes	27	4,500	100	180

In Big-O notation, retrieval time by key is:

In Big-O notation, retrieval time by key is:

- O(1) for `Hashtable`, `Dictionary`, and `OrderedDictionary`
- O(log *n*) for `SortedDictionary` and `SortedList`
- O(*n*) for `ListDictionary` (and nondictionary types such as `List<T>`)

where *n* is the number of elements in the collection.

IDictionary<TKey,TValue>

`IDictionary<TKey,TValue>` defines the standard protocol for all key/value-based col-

`IDictionary<TKey,TValue>` defines the standard protocol for all key/value-based collections. It extends `ICollection<T>` by adding methods and properties to access elements based on a key of arbitrary type:

```
public interface IDictionary <TKey, TValue> :
  ICollection <KeyValuePair <TKey, TValue>>, IEnumerable
{
   bool ContainsKey (TKey key);
   bool TryGetValue (TKey key, out TValue value);
   void Add         (TKey key, TValue value);
   bool Remove      (TKey key);

 }

TValue this [TKey key]      { get; set; }
ICollection <TKey> Keys     { get; }
ICollection <TValue> Values { get; }
```

```
// Main indexer - by key
// Returns just keys
// Returns just values
```

To add an item to a dictionary, you either call `Add` or use the index's set accessor—the latter adds an item to the dictionary if the key is not already present (or updates the item if it is present). Duplicate keys are forbidden in all dictionary implementations, so calling `Add` twice with the same key throws an exception.

To retrieve an item from a dictionary, use either the indexer or the `TryGetValue` method. If the key doesn't exist, the indexer throws an exception whereas `TryGetValue` returns `false`. You can test for membership explicitly by calling `ContainsKey`; however, this incurs the cost of two lookups if you then subsequently retrieve the item.

Enumerating directly over an `IDictionary<TKey,TValue>` returns a sequence of `KeyValuePair` structs:

KeyValuePair structs:

```
public struct KeyValuePair <TKey, TValue>
{
  public TKey Key     { get; }
  public TValue Value { get; }
}
```

You can enumerate over just the keys or values via the dictionary's `Keys`/`Values` properties.

We demonstrate the use of this interface with the generic `Dictionary` class in the following section.

IDictionary

The nongeneric `IDictionary` interface is the same in principle as `IDictionary<TKey,TValue>`, apart from two important functional differences. It's important to be aware of these differences, because `IDictionary` appears in legacy code (including the .NET Framework itself in places):

-
-

- Retrieving a nonexistent key via the indexer returns null (rather than throwing an exception).
- `Contains` tests for membership rather than `ContainsKey`.
- Enumerating over a nongeneric `IDictionary` returns a sequence of `DictionaryEntry` structs:

```
public struct DictionaryEntry
{
  public object Key   { get; set; }
  public object Value { get; set; }
}
```

Dictionary<TKey,TValue> and Hashtable

Dictionary<TKey,TValue> and Hashtable

The generic `Dictionary` class is one of the most commonly used collections (along with the `List<T>` collection). It uses a hashtable data structure to store keys and values, and it is fast and efficient.

The nongeneric version of `Dictionary<TKey,TValue>` is called `Hashtable`; there is no nongeneric class called `Dictionary`. When we refer simply to `Dictionary`, we mean the generic `Dictionary<TKey,TValue>` class.

`Dictionary` implements both the generic and nongeneric `IDictionary` interfaces, the generic `IDictionary` being exposed publicly. `Dictionary` is, in fact, a "textbook" implementation of the generic `IDictionary`.

Here's how to use it:

```
var d = new Dictionary<string, int>();
```

```
d.Add("One", 1);
d["Two"] = 2;
d["Two"] = 22;
d["Three"] = 3;

// adds to dictionary because "two" not already present
// updates dictionary because "two" is now present

Console.WriteLine (d["Two"]);
Console.WriteLine (d.ContainsKey ("One"));
Console.WriteLine (d.ContainsValue (3));
int val = 0;
if (!d.TryGetValue ("onE", out val))
```

```
if (!d.TryGetValue ("one", out val))
  Console.WriteLine ("No val");

// Prints "22"
// true (fast operation)
// true (slow operation)
// "No val" (case sensitive)

// Three different ways to enumerate the dictionary:

foreach (KeyValuePair<string, int> kv in d)
  Console.WriteLine (kv.Key + "; " + kv.Value);
foreach (string s in d.Keys) Console.Write (s);
Console.WriteLine();
foreach (int i in d.Values) Console.Write (i);
```

```
foreach (int i in d.Values) Console.Write (i);

// One ; 1
// Two ; 22
// Three ; 3
// OneTwoThree
// 1223
```


Its underlying hashtable works by converting each element's key into an integer hash code—a pseudounique value—and then applying an algorithm to convert the hash code into a hash key. This hash key is used internally to determine which "bucket" an entry belongs to. If the bucket contains more than one value, a linear search is performed on the bucket. A hashtable typically starts out maintaining a 1:1 ratio of buckets to values (a 1:1 *load factor*), meaning that each bucket contains only one

buckets to values (a 1:1 *load factor*), meaning that each bucket contains only one value. However, as more items are added to the hashtable, the load factor dynamically increases, in a manner designed to optimize insertion and retrieval performance as well as memory requirements.

A dictionary can work with keys of any type, providing it's able to determine equality between keys and obtain hash codes. By default, equality is determined via the key's `object.Equals` method, and the pseudounique hash code is obtained via the key's `GetHashCode` method. This behavior can be changed, either by overriding these methods or by providing an `IEqualityComparer` object when constructing the dictionary. A common application of this is to specify a case-insensitive equality comparer when using string keys:

```
var d = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

We discuss this further in "Plugging in Equality and Order" on page 304.

As with many other types of collections, the performance of a dictionary can be improved slightly by specifying the collection's expected size in the constructor, avoiding or lessening the need for internal resizing operations.

The nongeneric version is (more aptly) named `Hashtable` and is functionally similar apart from differences stemming from it exposing the nongeneric `IDictionary` in-

The nongeneric version is (more aptly) named `Hashtable` and is functionally similar apart from differences stemming from it exposing the nongeneric `IDictionary` interface discussed previously.

The downside to `Dictionary` and `Hashtable` is that the items are not sorted. Furthermore, the original order in which the items were added is not retained. As with all dictionaries, duplicate keys are not allowed.

OrderedDictionary

An `OrderedDictionary` is a nongeneric dictionary that maintains elements in the same order that they were added. With an `OrderedDictionary`, you can access elements both by index and by key.

An `OrderedDictionary` is not a *sorted* dictionary.

An `OrderedDictionary` is a combination of a `Hashtable` and an `ArrayList`. This means it has all the functionality of a `Hashtable`, plus functions such as `RemoveAt`, as well as an integer indexer. It also exposes `Keys` and `Values` properties that return elements in their original order.

This class was introduced in .NET 2.0, yet peculiarly, there's no generic version.

ListDictionary and HybridDictionary

`ListDictionary` uses a singly linked list to store the underlying data. It doesn't provide sorting, although it does preserve the original entry order of the items. `ListDictionary` is extremely slow with large lists. Its only real "claim to fame" is its efficiency with very small lists (fewer than 10 items).

`HybridDictionary` is a `ListDictionary` that automatically converts to a `Hashtable` upon reaching a certain size, to address `ListDictionary`'s problems with performance. The idea is to get a low memory footprint when the dictionary is small, and good performance when the dictionary is large. However, given the overhead in converting from one to the other—and the fact that a `Dictionary` is not excessively heavy or slow in either scenario—you wouldn't suffer unreasonably by using a `Dictionary` to begin with.

Both classes come only in nongeneric form.

Sorted Dictionaries

The Framework provides two dictionary classes internally structured such that their content is always sorted by key:

-
-

`SortedDictionary<TKey,TValue>`

`SortedList<TKey,TValue>`*

(In this section, we will abbreviate `<TKey,TValue>` to `<,>`.)

`SortedDictionary<,>` uses a red/black tree: a data structure designed to perform consistently well in any insertion or retrieval scenario.

`SortedList<,>` is implemented internally with an ordered array pair, providing fast

`SortedList<,>` is implemented internally with an ordered array pair, providing fast retrieval (via a binary-chop search) but poor insertion performance (because existing values have to be shifted to make room for a new entry).

`SortedDictionary<,>` is much faster than `SortedList<,>` at inserting elements in a random sequence (particularly with large lists). `SortedList<,>`, however, has an extra ability: to access items by index as well as by key. With a sorted list, you can go directly to the *n*th element in the sorting sequence (via the indexer on the `Keys`/`Values` properties). To do the same with a `SortedDictionary<,>`, you must manually enumerate over *n* items. (Alternatively, you could write a class that combines a sorted dictionary with a list class.)

None of the three collections allows duplicate keys (as is the case with all dictionaries).

The following example uses reflection to load all the methods defined in `System.Object` into a sorted list keyed by name, and then enumerates their keys and values:

```
// MethodInfo is in the System.Reflection namespace
```

* There's also a functionally identical nongeneric version of this called `SortedList`.

* There's also a functionally identical nongeneric version of this called `SortedList`.


```
var sorted = new SortedList <string, MethodInfo>();

foreach (MethodInfo m in typeof (object).GetMethods())
  sorted [m.Name] = m;

foreach (string name in sorted.Keys)
  Console.WriteLine (name);

foreach (MethodInfo m in sorted.Values)
  Console.WriteLine (m.Name + " returns a " + m.ReturnType);
```

Here's the result of the first enumeration:

```
Equals
GetHashCode
```

```
GetHashCode
GetType
ReferenceEquals
ToString
```

Here's the result of the second enumeration:

```
Equals returns a System.Boolean
GetHashCode returns a System.Int32
GetType returns a System.Type
ReferenceEquals returns a System.Boolean
ToString returns a System.String
```

Notice that we populated the dictionary through its indexer. If we instead used the `Add` method, it would throw an exception because the `object` class upon which we're reflecting overloads the `Equals` method, and you can't add the same key twice to a dictionary. By using the indexer, the later entry overwrites the earlier entry, preventing this error.

venting this error.

You can store multiple members of the same key by making each value element a list:

```
SortedList <string, List<MethodInfo>>
```

Extending our example, the following retrieves the `MethodInfo` whose key is `"GetHashCode"`, just as with an ordinary dictionary:

```
Console.WriteLine (sorted ["GetHashCode"]);      // Int32 GetHashCode()
```

So far, everything we've done would also work with a `SortedDictionary<,>`. The following two lines, however, which retrieve the last key and value, work only with a sorted list:

```
Console.WriteLine (sorted.Keys  [sorted.Count - 1]);
Console.WriteLine (sorted.Values[sorted.Count - 1].IsVirtual);
```

```
// ToString
// True
```


Customizable Collections and Proxies

The collection classes discussed in previous sections are convenient in that they can

The collection classes discussed in previous sections are convenient in that they can be directly instantiated, but they don't allow you to control what happens when an item is added to or removed from the collection. With strongly typed collections in an application, you sometimes need this control—for instance:

- To fire an event when an item is added or removed
- To update properties because of the added or removed item
- To detect an "illegal" add/remove operation and throw an exception (for example, if the operation violates a business rule)

The .NET Framework provides collection classes for this exact purpose, in the `System.Collections.ObjectModel` namespace. These are essentially proxies or wrappers that implement `IList<T>` or `IDictionary<,>` by forwarding the methods through to an underlying collection. Each `Add`, `Remove`, or `Clear` operation is routed via a virtual method that acts as a "gateway" when overridden.

Customizable collection classes are commonly used for publicly exposed collections;

Customizable collection classes are commonly used for publicly exposed collections; for instance, a collection of controls exposed publicly on a `System.Windows.Form` class.

Collection<T> and CollectionBase

`Collection<T>` class is a customizable wrapper for `List<T>`.

As well as implementing `IList<T>` and `IList`, it defines four additional virtual methods and a protected property as follows:

```
public class Collection<T> :
  IList<T>, ICollection<T>, IEnumerable<T>, IList, ICollection, IEnumerable
{
   // ...
```

```
protected virtual void ClearItems();
protected virtual void InsertItem (int index, T item);
protected virtual void RemoveItem (int index);
protected virtual void SetItem (int index, T item);
```

```
protected virtual void RemoveItem (int index);
protected virtual void SetItem (int index, T item);

protected IList<T> Items { get; }

}
```

The virtual methods provide the gateway by which you can "hook in" to change or enhance the list's normal behavior. The protected `Items` property allows the implementer to directly access the "inner list"—this is used to make changes internally without the virtual methods firing.

The virtual methods need not be overridden; they can be left alone until there's a requirement to alter the list's default behavior. The following example demonstrates the typical "skeleton" use of `Collection<T>`:


```
public class Animal
{
```

```
{
  public string Name;
  public int Popularity;
  public Animal (string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}
```

```
public class AnimalCollection : Collection <Animal>
{
  // AnimalCollection is already a fully functioning list of animals.
  // No extra code is required.
}
```

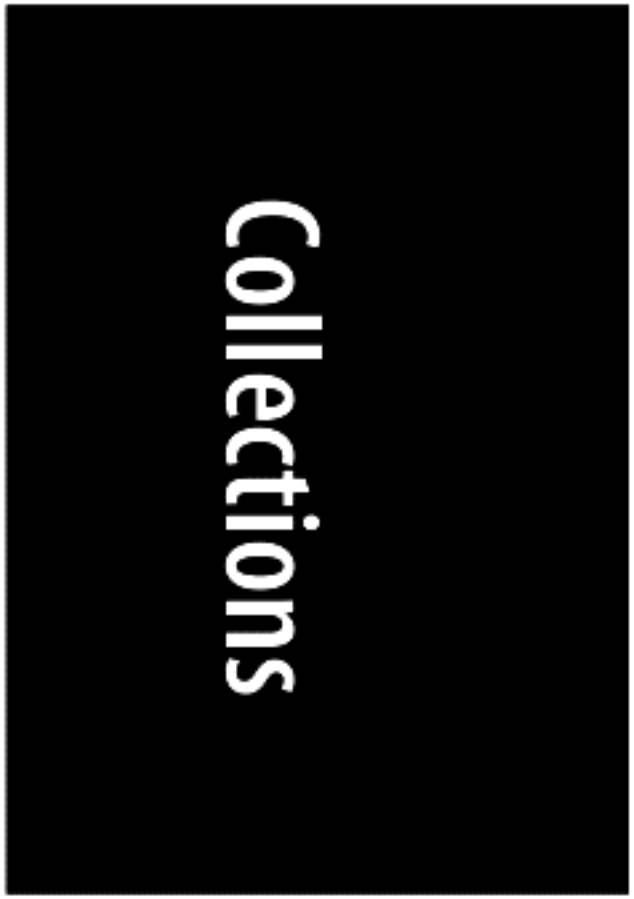
```
public class Zoo
{
```

```
  {
-
 // The class that will expose AnimalCollection.
 // This would typically have additional members.
-
   public readonly AnimalCollection Animals = new AnimalCollection();
}
-
class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
    zoo.Animals.Add (new Animal ("Kangaroo", 10));
    zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
    foreach (Animal a in zoo.Animals) Console.WriteLine (a.Name);
  }
}
-
```

As it stands, `AnimalCollection` is no more functional than a simple `List<Animal>`; its

As it stands, `AnimalCollection` is no more functional than a simple `List<Animal>`; its role is to provide a base for future extension. To illustrate, we'll now add a `Zoo` property to `Animal`, so it can reference the `Zoo` in which it lives and override each of the virtual methods in `Collection<Animal>` to maintain that property automatically:

```
public class Animal
{
  public string Name;
  public int Popularity;
  public Zoo Zoo { get; internal set; }
  public Animal(string name, int popularity)
  {
    Name = name; Popularity = popularity;
  }
}
```




```
public class AnimalCollection : Collection <Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
```

```
protected override void InsertItem (int index, Animal item)
{
  base.InsertItem (index, item);
  item.Zoo = zoo;
}
protected override void SetItem (int index, Animal item)
{
  base.SetItem (index, item);
  item.Zoo = zoo;
}
protected override void RemoveItem (int index)
{
  this [index].Zoo = null;
  base.RemoveItem (index);
}
protected override void ClearItems()
{
  foreach (Animal a in this) a.Zoo = null;
  base.ClearItems();
```

```
    foreach (Animal a in this) a.Zoo = null;
    base.ClearItems();
  }
}
```

```
public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}
```

`Collection<T>` also has a constructor accepting an existing `IList<T>`. Unlike with other collection classes, the supplied list is *proxied* rather than *copied*, meaning that subsequent changes will be reflected in the wrapping `Collection<T>` (although *without* `Collection<T>`'s virtual methods firing). Conversely, changes made via the `Collection<T>` will change the underlying list.

CollectionBase

`CollectionBase` is the nongeneric version of `Collection<T>` introduced in Framework

`CollectionBase` is the nongeneric version of `Collection<T>` introduced in Framework 1.0. This provides most of the same features as `Collection<T>` but is clumsier to use. Instead of the template methods `InsertItem`, `RemoveItem` `SetItem`, and `ClearItem`, `CollectionBase` has "hook" methods that double the number of methods required: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear`, and `OnClearComplete`. Because `CollectionBase` is nongeneric, you must also implement typed methods when subclassing it—at a minimum, a typed indexer and `Add` method.

KeyedCollection<TKey,TItem> and DictionaryBase

`KeyedCollection<TKey,TItem>` subclasses `Collection<TItem>`. It both adds and subtracts functionality. What it adds is the ability to access items by key, much like with a dictionary. What it subtracts is the ability to proxy your own inner list.

A keyed collection has some resemblance to an `OrderedDictionary` in that it combines a linear list with a hashtable. However, unlike `OrderedDictionary`, it doesn't implement `IDictionary` and doesn't support the concept of a key/value *pair*. Keys are obtained instead from the items themselves: via the abstract `GetKeyForItem`

are obtained instead from the items themselves: via the abstract `GetKeyForItem` method. This means enumerating a keyed collection is just like enumerating an ordinary list.

`KeyedCollection<TKey,TItem>` is best thought of as `Collection<TItem>` plus fast lookup by key.

Because it subclasses `Collection<>`, a keyed collection inherits all of `Collection<>`'s functionality, except for the ability to specify an existing list in construction. The additional members it defines are as follows:

```
public abstract class KeyedCollection <TKey, TItem> : Collection <TItem>

 // ...
 protected abstract TKey GetKeyForItem(TItem item);
 protected void ChangeItemKey(TItem item, TKey newKey);

// Fast lookup by key - this is in addition to lookup by index.
public TItem this[TKey key] { get; }

   protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

```
  protected IDictionary<TKey, TItem> Dictionary { get; }
}
```

`GetKeyForItem` is what the implementer overrides to obtain an item's key from the underlying object. The `ChangeItemKey` method must be called if the item's key property changes, in order to update the internal dictionary. The `Dictionary` property returns the internal dictionary used to implement the lookup, which is created when the first item is added. This behavior can be changed by specifying a creation threshold in the constructor, delaying the internal dictionary from being created until the threshold is reached (in the interim, a linear search is performed if an item is requested by key). A good reason not to specify a creation threshold is that having a valid dictionary can be useful in obtaining an `ICollection<>` of keys, via the `Dictionary`'s `Keys` property. This collection can then be passed on to a public property.

The most common use for `KeyedCollection<,>` is in providing a collection of items accessible both by index and by name. To demonstrate this, we'll revisit the zoo, this time implementing `AnimalCollection` as a `KeyedCollection<string,Animal>`:

```
public class Animal
{
  string name;
```

```
{
  string name;
  public string Name
  {
    get { return name; }
    set {
      if (Zoo != null) Zoo.Animals.NotifyNameChange (this, value);
      name = value;
    }
  }
```


```
    public int Popularity;
    public Zoo Zoo { get; internal set; }
    public Animal (string name, int popularity)
    {
      Name = name; Popularity = popularity;
    }
}
```

```
public class AnimalCollection : KeyedCollection <string, Animal>
{
  Zoo zoo;
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }
```

```
  public AnimalCollection (Zoo zoo) { this.zoo = zoo; }

internal void NotifyNameChange (Animal a, string newName)
{
  this.ChangeItemKey (a, newName);
}

protected override string GetKeyForItem (Animal item)
{
  return item.Name;
}

  // The following methods would be implemented as in the previous example
  protected override void InsertItem (int index, Animal item)...
  protected override void SetItem (int index, Animal item)...
  protected override void RemoveItem (int index)...
  protected override void ClearItems()...
}

public class Zoo
```

```
public class Zoo
{
  public readonly AnimalCollection Animals;
  public Zoo() { Animals = new AnimalCollection (this); }
}

class Program
{
  static void Main()
  {
    Zoo zoo = new Zoo();
    zoo.Animals.Add (new Animal ("Kangaroo", 10));
    zoo.Animals.Add (new Animal ("Mr Sea Lion", 20));
    Console.WriteLine (zoo.Animals [0].Popularity);
    Console.WriteLine (zoo.Animals ["Mr Sea Lion"].Popularity);
    zoo.Animals ["Kangaroo"].Name = "Mr Roo";
    Console.WriteLine (zoo.Animals ["Mr Roo"].Popularity);
  }
}
```

```
// 10
// 20
// 10
```

DictionaryBase

The nongeneric version of `KeyedCollection` is called `DictionaryBase`. This legacy class takes very different in its approach: it implements `IDictionary` and uses

clumsy hook methods like `CollectionBase`: `OnInsert`, `OnInsertComplete`, `OnSet`, `OnSetComplete`, `OnRemove`, `OnRemoveComplete`, `OnClear`, and `OnClearComplete` (and additionally, `OnGet`). The primary advantage of implementing `IDictionary` over taking the `KeyedCollection` approach is that you don't need to subclass it in order to obtain keys. But since the very purpose of `DictionaryBase` is to be subclassed, it's no advantage at all. The improved model in `KeyedCollection` is almost certainly due to

vantage at all. The improved model in `KeyedCollection` is almost certainly due to the fact that it was written some years later, with the benefit of hindsight. `DictionaryBase` is best considered useful for backward compatibility.

ReadOnlyCollection<T>

`ReadOnlyCollection<T>` is a wrapper, or *proxy*, that provides a read-only view of a collection. This is useful in allowing a class to publicly expose read-only access to a collection that the class can still update internally.

A read-only collection accepts the input collection in its constructor, to which it maintains a permanent reference. It doesn't take a static copy of the input collection, so subsequent changes to the input collection are visible through the read-only wrapper.

To illustrate, suppose your class wants to provide read-only public access to a list of strings called `Names`:

```
public class Test
{
```

```
public class Test
{
  public List<string> Names { get; private set; }
}
```

This only does half the job. Although other types cannot reassign the `Names` property, they can still call `Add`, `Remove`, or `Clear` on the list. The `ReadOnlyCollection<T>` class resolves this:

```
public class Test
{
  List<string> names;
  public ReadOnlyCollection<string> Names { get; private set; }

  public Test()
  {
    names = new List<string>();
    Names = new ReadOnlyCollection<string> (names);
  }

  public void AddInternally() { names.Add ("test"); }
```

```
  public void AddInternally() { names.Add ("test"); }
}
```


Now, only members within the `Test` class can alter the list of names:

```
Test t = new Test();
```

```
Console.WriteLine (t.Names.Count);
t.AddInternally();
Console.WriteLine (t.Names.Count);

// 0
// 1
```



```
t.Names.Add ("test");
((IList<string>) t.Names).Add ("test");
```

Plugging in Equality and Order

```
// Compiler error
// NotSupportedException
```

In the sections "Equality Comparison" on page 245 and "Order Comparison" on page 255 in Chapter 6, we described the standard .NET protocols that make a type equatable, hashable, and comparable. A type that implements these protocols can function correctly in a dictionary or sorted list "out of the box." More specifically:

-
-

A type for which `Equals` and `GetHashCode` return meaningful results can be used as a key in a `Dictionary` or `Hashtable`.

A type that implements `IComparable` /`IComparable<T>` can be used as a key in any of the *sorted* dictionaries or lists.

A type's default equating or comparison implementation typically reflects what is most "natural" for that type. Sometimes, however, the default behavior is not what

A type's default equating or comparison implementation typically reflects what is most "natural" for that type. Sometimes, however, the default behavior is not what you want. You might need a dictionary whose `string`-type key is treated case-insensitively. Or you might want a sorted list of customers, sorted by each customer's postcode. For this reason, the .NET Framework also defines a matching set of "plug-in" protocols. The plug-in protocols achieve two things:

-
-

They allow you to switch in alternative equating or comparison behavior.

They allow you to use a dictionary or sorted collection with a key type that's not intrinsically equatable or comparable.

The plug-in protocols consist of the following interfaces:

`IEqualityComparer` *and* `IEqualityComparer<T>`

- Performs plug-in *equality comparison and hashing*
- Recognized by `Hashtable` and `Dictionary`

`IComparer` *and* `IComparer<T>`

`IComparer` *and* `IComparer<T>`

- Performs plug-in *order comparison*
- Recognized by the sorted dictionaries and collections; also, `Array.Sort`

Each interface comes in generic and nongeneric forms. The `IEqualityComparer` interfaces also have a default implementation in a class called `EqualityComparer`.

In addition, Framework 4.0 adds two new interfaces called `IStructuralEquatable` and `IStructuralComparable`, which allow the option of structural comparisons on classes and arrays.

IEqualityComparer and EqualityComparer

An equality comparer switches in nondefault equality and hashing behavior, primarily for the `Dictionary` and `Hashtable` classes.

Recall the requirements of a hashtable-based dictionary. It needs answers to two questions for any given key:

questions for any given key:

-
-

Is it the same as another?

What is its integer hash code?

An equality comparer answers these questions for any given key:

-
-

1. `IEqualityComparer` interfaces:

by

implementing

the

```
public interface IEqualityComparer<T>
{
    bool Equals (T x, T y);
    int GetHashCode (T obj);
}
```

```
public interface IEqualityComparer
{
```

```
public interface IEqualityComparer
{
    bool Equals (object x, object y);
    int GetHashCode (object obj);
}

// Nongeneric version
```

To write a custom comparer, you implement one or both of these interfaces (implementing both gives maximum interoperability). As this is somewhat tedious, an alternative is to subclass the abstract `EqualityComparer` class, defined as follows:

```
public abstract class EqualityComparer<T> : IEqualityComparer,
                                            IEqualityComparer<T>
{
  public abstract bool Equals (T x, T y);
  public abstract int GetHashCode (T obj);
```

```
  public abstract int GetHashCode (T obj);

  bool IEqualityComparer.Equals (object x, object y);
  int IEqualityComparer.GetHashCode (object obj);

  public static EqualityComparer<T> Default { get; }
}
```

`EqualityComparer` implements both interfaces; your job is simply to override the two abstract methods.

The semantics for `Equals` and `GetHashCode` follow the same rules for `object.Equals` and `object.GetHashCode`, described in Chapter 6. In the following example, we define a `Customer` class with two fields, and then write an equality comparer that matches both the first and last names:

```
public class Customer
{
  public string LastName;
  public string FirstName;
```

```
    public string FirstName;
    public Customer (string last, string first)
    {
        LastName = last;
        FirstName = first;
    }
}
```

```
public class LastFirstEqComparer : EqualityComparer <Customer>
{
  public override bool Equals (Customer x, Customer y)
  {
    return x.LastName == y.LastName && x.FirstName == y.FirstName;
  }

  public override int GetHashCode (Customer obj)
  {
    return (obj.LastName + ";" + obj.FirstName).GetHashCode();
  }
}
```

To illustrate how this works, we'll create two customers:

To illustrate how this works, we'll create two customers:

```
Customer c1 = new Customer ("Bloggs", "Joe");
Customer c2 = new Customer ("Bloggs", "Joe");
```

Because we haven't overridden `object.Equals`, normal reference type equality semantics apply:

```
Console.WriteLine (c1 == c2);              // False
Console.WriteLine (c1.Equals (c2));        // False
```

The same default equality semantics apply when using these customers in a `Dictionary` without specifying an equality comparer:

```
var d = new Dictionary<Customer, string>();
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));        // False
```

Now with the custom equality comparer:

```
var eqComparer = new LastFirstEqComparer();
var d = new Dictionary<Customer, string> (eqComparer);
d [c1] = "Joe";
```

```
d [c1] = "Joe";
Console.WriteLine (d.ContainsKey (c2));          // True
```

In this example, we would have to be careful not to change the customer's `FirstName` or `LastName` while it was in use in the dictionary. Otherwise, its hash code would change and the `Dictionary` would break.

EqualityComparer<T>.Default

Calling `EqualityComparer<T>.Default` returns a general-purpose equality comparer that can be used as an alternative to the static `object.Equals` method. The advantage is that first checks if `T` implements `IEquatable<T>` and if so, calls that implementation instead, avoiding the boxing overhead. This is particularly useful in generic methods:

```
static bool Foo<T> (T x, T y)
 {
   bool same = EqualityComparer<T>.Default.Equals (x, y);
   ...
```


IComparable and IComparer

IComparer and Comparer

Comparers are used to switch in custom ordering logic for sorted dictionaries and collections.

Note that a comparer is useless to the unsorted dictionaries such as `Dictionary` and `Hashtable`—these require an `IEqualityComparer` to get hash codes. Similarly, an equality comparer is useless for sorted dictionaries and collections.

Here are the `IComparer` interface definitions:

```
public interface IComparer
{
  int Compare(object x, object y);
}
public interface IComparer <in T>
```

```
public interface IComparer <in T>
{
   int Compare(T x, T y);
}
```

As with equality comparers, there's an abstract class you can subtype instead of implementing the interfaces:

```
public abstract class Comparer<T> : IComparer, IComparer<T>
{
   public static Comparer<T> Default { get; }

   public abstract int Compare (T x, T y);
   int IComparer.Compare (object x, object y);
}
```

```
int IComparer.Compare (object x, object y);

// Implemented by you
// Implemented for you
```

The following example illustrates a class that describes a wish, and a comparer that sorts wishes by priority:

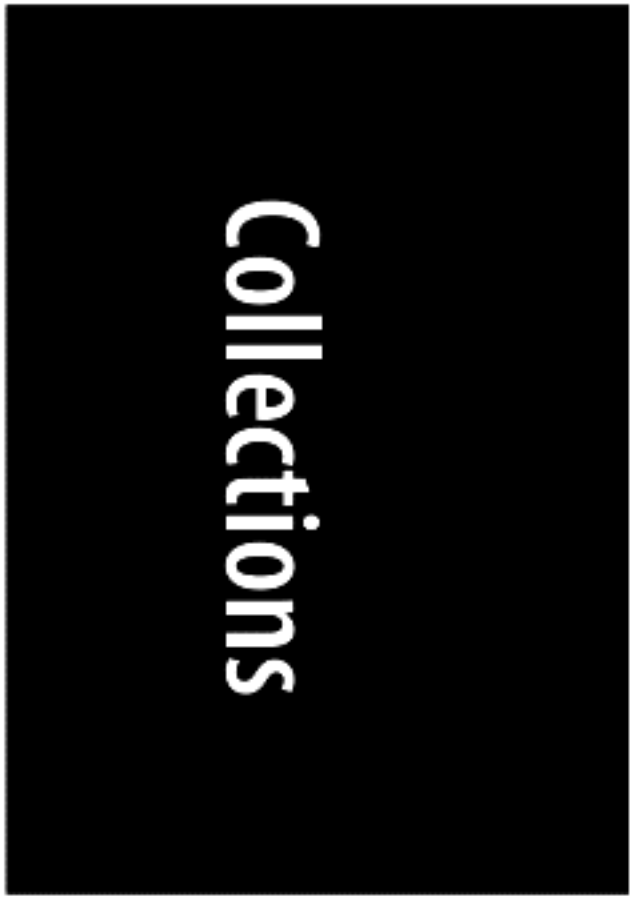
```
class Wish
{
  public string Name;
  public int Priority;

  public Wish (string name, int priority)
  {
```

```
    {
      Name = name;
      Priority = priority;
    }
  }
  class PriorityComparer : Comparer <Wish>
  {
    public override int Compare (Wish x, Wish y)
    {
      if (object.Equals (x, y)) return 0;
      return x.Priority.CompareTo (y.Priority);
    }
  }
```

```
// Fail-safe check
```

```
// Fail safe check
```



The `object.Equals` check ensures that we can never contradict the `Equals` method. Calling the static `object.Equals` method in this case is better than calling `x.Equals` because it still works if `x` is null!

Here's how our PriorityComparer is used to sort a `List`:

Here's how our PriorityComparer is used to sort a List:

```
var wishList = new List<Wish>();
wishList.Add (new Wish ("Peace", 2));
wishList.Add (new Wish ("Wealth", 3));
wishList.Add (new Wish ("Love", 2));
wishList.Add (new Wish ("3 more wishes", 1));

wishList.Sort (new PriorityComparer());
foreach (Wish w in wishList) Console.Write (w.Name + "
| ");

// OUTPUT: 3 more wishes
| Love | Peace | Wealth |
```

In the next example, SurnameComparer allows you to sort surname strings in an order suitable for a phonebook listing:

```
class SurnameComparer : Comparer <string>
{
  string Normalize (string s)
  {
    s = s.Trim().ToUpper();
    if (s.StartsWith ("MC")) s = "MAC" + s.Substring (2);
    return s;
  }

  public override int Compare (string x, string y)
  {
    return Normalize (x).CompareTo (Normalize (y));
  }
}
```

Here's SurnameComparer in use in a sorted dictionary:

```
var dic = new SortedDictionary<string,string> (new SurnameComparer());
dic.Add ("MacPhail", "second!");
dic.Add ("MacWilliam", "third!");
dic.Add ("McDonald", "first!");

foreach (string s in dic.Values)
  Console.Write (s + " ");

// first! second! third!
```

StringComparer

StringComparer is a predefined plug-in class for equating and comparing strings, allowing you to specify language and case sensitivity. StringComparer implements both IEqualityComparer and IComparer (and their generic versions), so it can be used

both `IEqualityComparer` and `IComparer` (and their generic versions), so it can be used with any type of dictionary or sorted collection:

```
    // CultureInfo is defined in System.Globalization

public abstract class StringComparer : IComparer, IComparer <string>,
                                       IEqualityComparer,
                                       IEqualityComparer <string>

{
  public abstract int Compare (string x, string y);
  public abstract bool Equals (string x, string y);
  public abstract int GetHashCode (string obj);

  public static StringComparer Create (CultureInfo culture,
                                       bool ignoreCase);
  public static StringComparer CurrentCulture { get; }
  public static StringComparer CurrentCultureIgnoreCase { get; }
  public static StringComparer InvariantCulture { get; }
```

```
  public static StringComparer CurrentCultureIgnoreCase { get; }
  public static StringComparer InvariantCulture { get; }
  public static StringComparer InvariantCultureIgnoreCase { get; }
  public static StringComparer Ordinal { get; }
  public static StringComparer OrdinalIgnoreCase { get; }
}
```

Because `StringComparer` is abstract, you obtain instances via its static methods and properties. `StringComparer.Ordinal` mirrors the default behavior for string equality comparison and `StringComparer.CurrentCulture` for order comparison.

In the following example, an ordinal case-insensitive dictionary is created, such that `dict["Joe"]` and `dict["JOE"]` mean the same thing:

```
var dict = new Dictionary<string, int> (StringComparer.OrdinalIgnoreCase);
```

In the next example, an array of names is sorted, using Australian English:

```
string[] names = { "Tom", "HARRY", "sheila" };
CultureInfo ci = new CultureInfo ("en-AU");
Array.Sort<string> (names, StringComparer.Create (ci, false));
```

The final example is a culture-aware version of the `SurnameComparer` we wrote in the previous section (to compare names suitable for a phonebook listing):

The final example is a culture-aware version of the `SurnameComparer` we wrote in the previous section (to compare names suitable for a phonebook listing):

```
class SurnameComparer : Comparer <string>
{
  StringComparer strCmp;

public SurnameComparer (CultureInfo ci)
{
  // Create a case-sensitive, culture-sensitive string comparer
  strCmp = StringComparer.Create (ci, false);
}

string Normalize (string s)
{
  s = s.Trim();
  if (s.ToUpper().StartsWith ("MC")) s = "MAC" + s.Substring (2);
  return s;
}
```

```
        return strCmp.Compare (Normalize (x), Normalize (y));
    }
    // Directly call Compare on our culture-aware StringComparer
    public override int Compare (string x, string y)
    {
```

Collections

```
}
```


IStructuralEquatable and IStructuralComparable

As we said in the previous chapter, structs implement *structural comparison* by default: two structs are equal if all of their fields are equal. Sometimes, however, structural equality and order comparison are useful as plug-in options on other types as well—such as arrays and tuples. Framework 4.0 introduces two new interfaces to help with this:

```
public interface IStructuralEquatable
{
  bool Equals (object other, IEqualityComparer comparer);
  int GetHashCode (IEqualityComparer comparer);
}

public interface IStructuralComparable
{
```

```
public interface IStructuralComparable
{
  int CompareTo (object other, IComparer comparer);
}
```

The `IEqualityComparer/IComparer` that you pass in are applied to each individual element in the composite object. We can demonstrate this using arrays and tuples, which implement these interfaces: in the following example, we compare two arrays for equality: first using the default `Equals` method, and then using `IStructuralEquatable`'s version:

```
int[] a1 = { 1, 2, 3 };
int[] a2 = { 1, 2, 3 };
Console.Write (a1.Equals (a2));
Console.Write (a1.Equals (a2, EqualityComparer<int>.Default));
```

```
// False
// True
```

Here's another example:

Here's another example:

```
string[] a1 = "the quick brown fox".Split();
string[] a2 = "THE QUICK BROWN FOX".Split();
bool isTrue = a1.Equals (a2, StringComparer.InvariantCultureIgnoreCase);
```

Tuples work in the same way:

```
var t1 = Tuple.Create (1, "foo");
var t2 = Tuple.Create (1, "FOO");
bool isTrue = t1.Equals (t2, StringComparer.InvariantCultureIgnoreCase);
int zero = t1.CompareTo (t2, StringComparer.InvariantCultureIgnoreCase);
```

The difference with tuples, though, is that their *default* equality and order comparison implementations also apply structural comparisons:

```
var t1 = Tuple.Create (1, "FOO");
var t2 = Tuple.Create (1, "FOO");
Console.WriteLine (t1.Equals (t2));   // True
```


8

LINQ Queries

LINQ, or Language Integrated Query, is a set of language and framework features for writing structured type-safe queries over local object collections and remote data sources. LINQ was introduced in C# 3.0 and Framework 3.5.

LINQ enables you to query any collection implementing `IEnumerable<T>`, whether an array, list, or XML DOM, as well as remote data sources, such as tables in SQL Server. LINQ offers the benefits of both compile-time type checking and dynamic query composition.

Server. LINQ offers the benefits of both compile-time type checking and dynamic query composition.

This chapter describes the LINQ architecture and the fundamentals of writing queries. All core types are defined in the `System.Linq` and `System.Linq.Expressions` namespaces.

The examples in this and the following two chapters are preloaded into an interactive querying tool called LINQPad. You can download LINQPad from *www.linqpad.net*.

Getting Started

The basic units of data in LINQ are *sequences* and *elements*. A sequence is any object that implements `IEnumerable<T>` and an element is each item in the sequence. In the following example, `names` is a sequence, and `Tom`, `Dick`, and `Harry` are elements:

```
string[] names = { "Tom", "Dick", "Harry" };
```

We call this a *local sequence* because it represents a local collection of objects in

We call this a *local sequence* because it represents a local collection of objects in memory.

A *query operator* is a method that transforms a sequence. A typical query operator accepts an *input sequence* and emits a transformed *output sequence*. In the `Enumerable` class in `System.Linq`, there are around 40 query operators—all implemented as static extension methods. These are called *standard query operators*.

Queries that operate over local sequences are called local queries or *LINQ-to-objects* queries.

LINQ also supports sequences that can be dynamically fed from a remote data source such as a SQL Server. These sequences additionally implement the `IQueryable<T>` interface and are supported through a matching set of standard query operators in the `Queryable` class. We discuss this further in the section "Interpreted Queries" on page 339 later in this chapter.

Interpreted Queries" on page 889 later in this chapter.

A query is an expression that transforms sequences with query operators. The simplest query comprises one input sequence and one operator. For instance, we can apply the `Where` operator on a simple array to extract those whose length is at least four characters as follows:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine (n);
```

```
Dick
Harry
```

Because the standard query operators are implemented as extension methods, we can call `Where` directly on `names`—as though it were an instance method:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

For this to compile, you must import the `System.Linq` namespace. Here's a complete

For this to compile, you must import the `System.Linq` namespace. Here's a complete example:

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

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry" };

    IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
    foreach (string name in filteredNames) Console.WriteLine (name);
  }
}
```

```
  }
}
```

```
Dick
Harry
```


We could further shorten our code by implicitly typing `filteredNames`:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

This can hinder readability, however, particularly outside of an IDE, where there are no tool tips to help.

In this chapter, we avoid implicitly typing query results except when it's mandatory (as we'll see later, in the section "Projection Strategies" on page 337), or when a query's type is irrelevant