Order Comparison

As well as defining standard protocols for equality, C# and .NET define standard protocols for determining the order of one object relative to another. The basic protocols are:

-
-

The IComparable interfaces (IComparable and IComparable<T>)

The > and < operators

The IComparable interfaces are used by general-purpose sorting algorithms. In the following example, the static Array.Sort method works because System.String implements the IComparable interfaces:

```
string[] colors = { "Green", "Red", "Blue" };
Array.Sort (colors);
foreach (string c in colors) Console.Write (c + " ");   // Blue Green Red
```

The < and > operators are more specialized, and they are intended mostly for numeric types. Because they are statically resolved, they can translate to highly efficient bytecode, suitable for computationally intensive algorithms.

The .NET Framework also provides pluggable ordering protocols, via the IComparer interfaces. We describe these in the final section of Chapter 7.

`IComparer` interfaces. We describe these in the final section of Chapter 7.

IComparable

The `IComparable` interfaces are defined as follows:

```
public interface IComparable       { int CompareTo (object other); }
public interface IComparable<in T> { int CompareTo (T other);      }
```

The two interfaces represent the same functionality. With value types, the generic type-safe interface is faster than the nongeneric interface. In both cases, the `CompareTo` method works as follows:

-
-
-

If `a` comes after `b`, `a.CompareTo(b)` returns a positive number.

If `a` is the same as `b`, `a.CompareTo(b)` returns `0`.

If `a` comes before `b`, `a.CompareTo(b)` returns a negative number.

If a comes before b, `a.CompareTo(b)` returns a negative number.

For example:

```
Console.WriteLine ("Beck".CompareTo ("Anne"));
Console.WriteLine ("Beck".CompareTo ("Beck"));
Console.WriteLine ("Beck".CompareTo ("Chris"));
```

```
// 1
// 0
// -1
```

Most of the base types implement both `IComparable` interfaces. These interfaces are also sometimes implemented when writing custom types. An example is given shortly.

IComparable versus Equals

IComparable versus Equals

Consider a type that both overrides `Equals` and implements the `IComparable` interfaces. You'd expect that when `Equals` returns `true`, `CompareTo` should return `0`. And you'd be right. But here's the catch:

> When `Equals` returns `false`, `CompareTo` can return what it likes!

In other words, equality can be "fussier" than comparison, but not vice versa (violate this and sorting algorithms will break). So, `CompareTo` can say "All objects are equal" while `Equals` says "But some are more equal than others!"

A great example of this is `System.String`. `String`'s `Equals` method and `==` operator use *ordinal* comparison, which compares the Unicode point values of each character. Its `CompareTo` method, however, uses a less fussy *culture-dependent* comparison. On most computers, for instance, the strings "ǔ" and "ǖ" are different according to `Equals`, but the same according to `CompareTo`.

In Chapter 7, we discuss the pluggable ordering protocol, `IComparer`, which allows you to specify an alternative ordering algorithm when sorting or instantiating a sorted collection. A custom `IComparer` can further extend the gap between `CompareTo` and `Equals`—a case-insensitive string comparer, for instance, will return `0` when

and `Equals`—a case-insensitive string comparer, for instance, will return `0` when comparing `"A"` and `"a"`. The reverse rule still applies, however: `CompareTo` can never be fussier than `Equals`.

When implementing the `IComparable` interfaces in a custom type, you can avoid running afoul of this rule by writing the first line of `CompareTo` as follows:

```
if (Equals (other)) return 0;
```

After that, it can return what it likes, as long as it's consistent!

< and >

Some types define < and > operators. For instance:

```
bool after2010 = DateTime.Now > new DateTime (2010, 1, 1);
```

```
bool after2010 = DateTime.Now > new DateTime (2010, 1, 1);
```

When implemented, the < and > operators are functionally consistent with the `IComparable` interfaces. This is standard practice across the .NET Framework.

It's also standard practice to implement the `IComparable` interfaces whenever < and > are overloaded, although the reverse is not true. In fact, most .NET types that implement `IComparable` *do not* overload < and >. This differs from the situation with equality, where it's normal to overload == when overriding `Equals`.

Typically, > and < are overloaded only when:

-
-
-

A type has a strong intrinsic concept of "greater than" and "less than" (versus `IComparable`'s broader concepts of "comes before" and "comes after").

There is only one way *or context* in which to perform the comparison.

The result is invariant across cultures.

The result is invariant across cultures.

`System.String` doesn't satisfy the last point: the results of string comparisons can vary according to language. Hence, `string` doesn't support the > and < operators:

```
bool error = "Beck" > "Anne";        // Compile-time error
```

Implementing the IComparable Interfaces

In the following struct, representing a musical note, we implement the `IComparable` interfaces, as well as overloading the < and > operators. For completeness, we also override `Equals`/`GetHashCode` and overload == and !=:

```
public struct Note : IComparable<Note>, IEquatable<Note>, IComparable
{
  int _semitonesFromA;
  public int SemitonesFromA { get { return _semitonesFromA; } }
```

```
  public Note (int semitonesFromA)
  {
    _semitonesFromA = semitonesFromA;
  }
  public int CompareTo (Note other)
  {
```

```
  {
```

```
  // Generic IComparable<T>
```



```
    if (Equals (other)) return 0;     // Fail-safe check
    return _semitonesFromA.CompareTo (other._semitonesFromA);
  }

  int IComparable.CompareTo (object other)     // Nongeneric IComparable
  {
    if (!(other is Note))
      throw new InvalidOperationException ("CompareTo: Not a note");
    return CompareTo ((Note) other);
  }

  public static bool operator < (Note n1, Note n2)
  {
```

```
{
    return n1.CompareTo (n2) < 0;
}
public static bool operator > (Note n1, Note n2)
{
  return n1.CompareTo (n2) > 0;
}

public bool Equals (Note other)    // for IEquatable<Note>
{
  return _semitonesFromA == other._semitonesFromA;
}

  public override bool Equals (object other)
  {
    if (!(other is Note)) return false;
    return Equals ((Note) other);
```

```
    return Equals ((Note) other);
  }
  public override int GetHashCode()
  {
    return _semitonesFromA.GetHashCode();
  }
  public static bool operator == (Note n1, Note n2)
  {
     return n1.Equals (n2);
  }
  public static bool operator != (Note n1, Note n2)
  {
     return !(n1 == n2);
  }
}
```

Utility Classes

Console

The static `Console` class handles standard input/output for console-based applications. In a command-line (Console) application, the input comes from the keyboard

via `Read`, `ReadKey`, and `ReadLine`, and the output goes to the text window via `Write` and `WriteLine`. You can control the window's position and dimensions with the properties `WindowLeft`, `WindowTop`, `WindowHeight`, and `WindowWidth`. You can also change the `BackgroundColor` and `ForegroundColor` properties and manipulate the cursor with the `CursorLeft`, `CursorTop`, and `CursorSize` properties:

```
Console.WindowWidth = Console.LargestWindowWidth;
Console.ForegroundColor = ConsoleColor.Green;
```

```
Console.ForegroundColor = ConsoleColor.Green;
Console.Write ("test... 50%");
Console.CursorLeft -= 3;
Console.Write ("90%");     // test... 90%
```

The `Write` and `WriteLine` methods are overloaded to accept a composite format string (see `String.Format` in "String and Text Handling" on page 193). However, neither method accepts a format provider, so you are stuck with `CultureInfo.CurrentCulture`. (The workaround, of course, is to explicitly call `string.Format`.)

The `Console.Out` property returns a `TextWriter`. Passing `Console.Out` to a method that expects a `TextWriter` is a useful way to get that method to write to the `Console` for diagnostic purposes.

You can also redirect the `Console`'s input and output streams via the `SetIn` and `SetOut` methods:

```
// First save existing output writer:
System.IO.TextWriter oldOut = Console.Out;
```

```
System.IO.TextWriter oldOut = Console.Out;

// Redirect the console's output to a file:
using (System.IO.TextWriter w = System.IO.File.CreateText
                                ("e:\\output.txt"))
{
  Console.SetOut (w);
  Console.WriteLine ("Hello world");
}

// Restore standard console output
Console.SetOut (oldOut);

// Open the output.txt file in Notepad:
System.Diagnostics.Process.Start ("e:\\output.txt");
```

In Chapter 14, we describe how streams and text writers work.

In a Visual Studio Windows application, the `Console`'s output is automatically redirected to Visual Studio's output window (in debug mode). This can make `Console.Write` useful for diagnostic purposes; although in most cases the `Debug` and `Trace` classes in the `System.Diagnostics` namespace are more appropriate (see Chapter 13).

Environment

The static `System.Environment` class provides a range of useful properties:

Files and folders

`CurrentDirectory`, `SystemDirectory`, `CommandLine`

Computer and operating system

`MachineName`, `ProcessorCount`, `OSVersion`

User logon

`UserName`, `UserInteractive`, `UserDomainName`

Diagnostics

`TickCount`, `StackTrace`, `WorkingSet`, `Version`

You can obtain additional folders by calling `GetFolderPath`; we describe this in "File and Directory Operations" on page 559 in Chapter 14.

You can access OS environment variables (what you see when you type "set" at the command prompt) with the following three methods: `GetEnvironmentVariable`, `GetEnvironmentVariables`, and `SetEnvironmentVariable`.

The `ExitCode` property lets you set the return code, for when your program is called from a command or batch file, and the `FailFast` method terminates a program immediately, without performing cleanup.

Process

The `Process` class in `System.Diagnostics` allows you to launch a new process.

The static `Process.Start` method has a number of overloads; the simplest accepts a simple filename with optional arguments:

```
Process.Start ("notepad.exe");
Process.Start ("notepad.exe", "e:\\file.txt");
```

You can also specify just a filename, and the registered program for its extension will be launched:

```
Process.Start ("e:\\file.txt");
```

The most flexible overload accepts a `ProcessStartInfo` instance. With this, you can capture and redirect the launched process's input, output, and error output (if you set `UseShellExecute` to `false`). The following captures the output of calling `ipconfig`:

```
ProcessStartInfo psi = new ProcessStartInfo
{
  FileName = "cmd.exe",
  Arguments = "/c ipconfig /all",
  RedirectStandardOutput = true,
  UseShellExecute = false
};
Process p = Process.Start (psi);
```

```
};
Process p = Process.Start (psi);
string result = p.StandardOutput.ReadToEnd();
Console.WriteLine (result);
```


You can do the same to invoke the `csc` compiler, if you set `Filename` to the following:

```
psi.FileName = System.IO.Path.Combine (
  System.Runtime.InteropServices.RuntimeEnvironment.GetRuntimeDirectory(),
  "csc.exe");
```

If you don't redirect output, `Process.Start` executes the program in parallel to the caller. If you want to wait for the new process to complete, you can call `WaitForExit` on the `Process` object, with an optional timeout.

The `Process` class also allows you to query and interact with other processes running on the computer (see Chapter 13).

FW Fundamentals

7

7

Collections

The .NET Framework provides a standard set of types for storing and managing collections of objects. These include resizable lists, linked lists, and sorted and unsorted dictionaries, as well as arrays. Of these, only arrays form part of the C# language; the remaining collections are just classes you instantiate like any other.

The types in the Framework for collections can be divided into the following categories:

-
-
-

Interfaces that define standard collection protocols

Interfaces that define standard collection protocols

Ready-to-use collection classes (lists, dictionaries, etc.)

Base classes for writing application-specific collections

This chapter covers each of these categories, with an additional section on the types used in determining element equality and order.

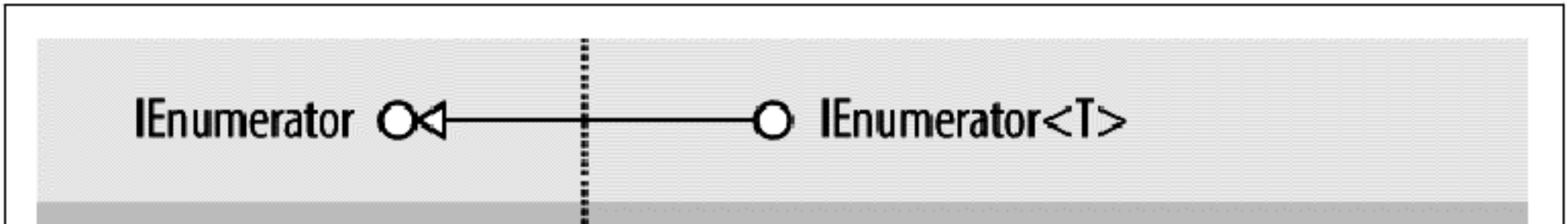
The collection namespaces are as follows:

Namespace	Contains
`System.Collections`	Nongeneric collection classes and interfaces
`System.Collections.Specialized`	Strongly typed nongeneric collection classes
`System.Collections.Generic`	Generic collection classes and interfaces
`System.Collections.ObjectModel`	Proxies and bases for custom collections
`System.Collections.Concurrent`	Thread-safe collections (see Chapter 22)

Enumeration

In computing, there are many different kinds of collections ranging from simple data structures, such as arrays or linked lists, to more complex ones, such as red/black trees and hashtables. Although the internal implementation and external characteristics of these data structures vary widely, the ability to traverse the contents of the collection is an almost universal need. The Framework supports this need via a

pair of interfaces (`IEnumerable`, `IEnumerator`, and their generic counterparts) that allow different data structures to expose a common traversal API. These are part of a larger set of collection interfaces illustrated in Figure 7-1.



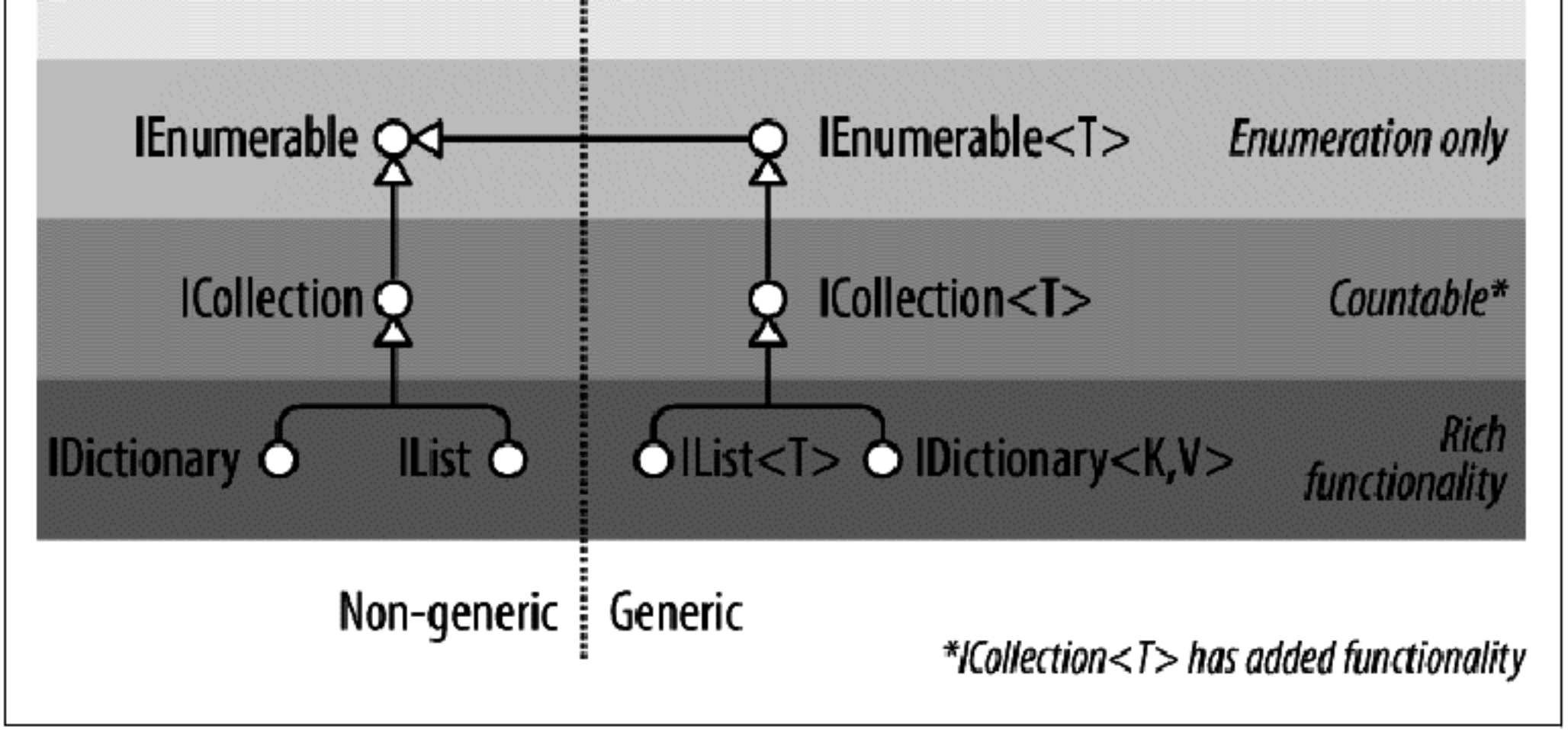


Figure 7-1. Collection interfaces

IEnumerable and IEnumerator

The `IEnumerator` interface defines the basic low-level protocol by which elements in a collection are traversed—or enumerated—in a forward-only manner. Its declaration is as follows:

tion is as follows:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
  void Reset();
}
```

`MoveNext` advances the current element or "cursor" to the next position, returning `false` if there are no more elements in the collection. `Current` returns the element at the current position (usually cast from `object` to a more specific type). `MoveNext` must be called before retrieving the first element—this is to allow for an empty collection. The `Reset` method, if implemented, moves back to the start, allowing the collection to be enumerated again. (Calling `Reset` is generally avoided because it's not supported by all enumerators.)

ported by all enumerators.)

Collections do not *implement* enumerators; instead, they *provide* enumerators, via the interface `IEnumerable`:

```
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```


By defining a single method retuning an enumerator, `IEnumerable` provides flexibility in that the iteration logic can be farmed off to another class. Moreover, it means that several consumers can enumerate the collection at once without interfering with each other. `IEnumerable` can be thought of as "IEnumeratorProvider," and it is the most basic interface that collection classes implement.

The following example illustrates low-level use of `IEnumerable` and `IEnumerator`:

The following example illustrates low-level use of `IEnumerable` and `IEnumerator`:

```
string s = "Hello";

// Because string implements IEnumerable, we can call GetEnumerator():
IEnumerator rator = s.GetEnumerator();

while (rator.MoveNext())
{
  char c = (char) rator.Current;
  Console.Write (c + ".");
}
// Output:  H.e.l.l.o.
```

However, it's rare to call methods on enumerators directly in this manner, because C# provides a syntactic shortcut: the `foreach` statement. Here's the same example rewritten using `foreach`:

```
string s = "Hello";      // The String class implements IEnumerable

foreach (char c in s)
  Console.Write (c + ".");
```

IEnumerable<T> and IEnumerator<T>

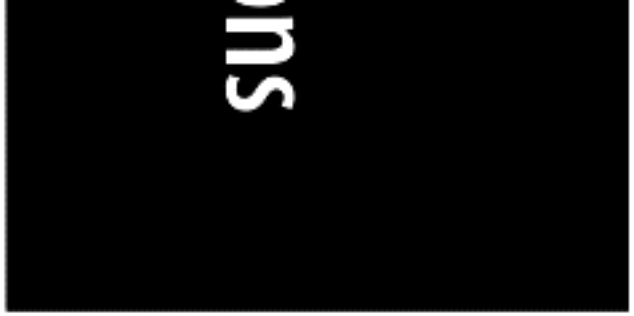
`IEnumerator` and `IEnumerable` are nearly always implemented in conjunction with their extended generic versions:

```
public interface IEnumerator<T> : IEnumerator, IDisposable
{
  T Current { get; }
}
```

```
public interface IEnumerable<T> : IEnumerable
```

```
public interface IEnumerable<T> : IEnumerable
{
  IEnumerator<T> GetEnumerator();
}
```

By defining a typed version of `Current` and `GetEnumerator`, these interfaces strengthen static type safety, avoid the overhead of boxing with value-type elements, and are more convenient to the consumer. Arrays automatically implement `IEnumerable<T>` (where `T` is the member type of the array).



Thanks to the improved static type safety, calling the following method with an array of characters will generate a compile-time error:

```
void Test (IEnumerable<int> numbers) { ... }
```


It's a standard practice for collection classes to publicly expose `IEnumerable<T>`, while "hiding" the nongeneric `IEnumerable` through explicit interface implementation. This is so that if you directly call `GetEnumerator()`, you get back the type-safe generic `IEnumerator<T>`. There are times, though, when this rule is broken for reasons of backward compatibility (generics did not exist prior to C# 2.0). A good example is arrays—these must return the nongeneric (the nice way of putting it is "classic") `IEnumerator` to avoid breaking earlier code. In order to get a generic `IEnumerator<T>`, you must cast to expose the explicit interface:

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

```
int[] data = { 1, 2, 3 };
var rator = ((IEnumerable <int>)data).GetEnumerator();
```

Fortunately, you rarely need to write this sort of code, thanks to the `foreach` statement.

IEnumerable<T> and IDisposable

IEnumerable<T> implements `IDisposable`. This allows enumerators to hold references to resources such as database connections—and ensure that those resources are released when enumeration is complete (or abandoned partway through). The `foreach` statement recognizes this detail and translates this:

```
foreach (var element in somethingEnumerable) { ... }
```

into this:

```
using (var rator = somethingEnumerable.GetEnumerator())
  while (rator.MoveNext())
  {
    var element = rator.Current;

    ...
```

```
    ...
}
```

The `using` block ensures disposal—more on `IDisposable` in Chapter 12.

Implementing the Enumeration Interfaces

You might want to implement `IEnumerable` or `IEnumerable<T>` for one or more of the following reasons:

-
-
-
-

To support the `foreach` statement

To interoperate with anything expecting a standard collection

As part of implementing a more sophisticated collection interface

As part of implementing a more sophisticated collection interface

To support collection initializers

To implement `IEnumerable`/`IEnumerable<T>`, you must provide an enumerator. You can do this in one of three ways:

-
-
-

If the class is "wrapping" another collection, by returning the wrapped collection's enumerator

Via an iterator using `yield return`

By instantiating your own `IEnumerator`/`IEnumerator<T>` implementation

You can also subclass an existing collection: `Collection<T>` is

You can also subclass an existing collection: `Collection<T>` is designed just for this purpose (see "Customizable Collections and Proxies" on page 298). Yet another approach is to use the LINQ query operators that we'll cover in the next chapter.

Returning another collection's enumerator is just a matter of calling `GetEnumerator` on the inner collection. However, this is viable only in the simplest scenarios, where the items in the inner collection are exactly what are required. A more flexible approach is to write an iterator, using C#'s `yield return` statement. An *iterator* is a C# language feature that assists in writing collections, in the same way the `foreach` statement assists in consuming collections. An iterator automatically handles the implementation of `IEnumerable` and `IEnumerator`—or their generic versions. Here's a simple example:

```
public class MyCollection : IEnumerable
{
  int[] data = { 1, 2, 3 };

  public IEnumerator GetEnumerator()
```

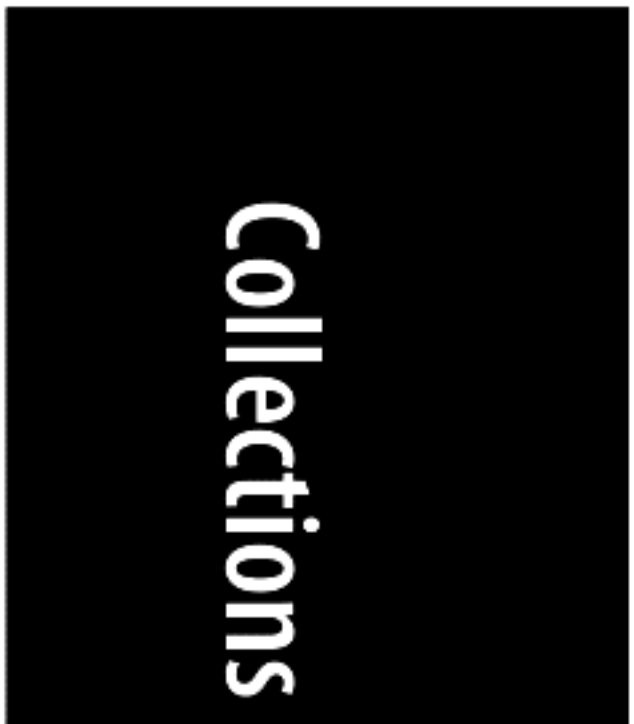
```
    public IEnumerator GetEnumerator()
    {
      foreach (int i in data)
        yield return i;
    }
  }
```

Notice the "black magic": `GetEnumerator` doesn't appear to return an enumerator at all! Upon parsing the `yield return` statement, the compiler writes a hidden nested enumerator class behind the scenes, and then refactors `GetEnumerator` to instantiate and return that class. Iterators are powerful and simple (and are the basis for LINQ's implementation).

Keeping with this approach, we can also implement the generic interface `IEnumerable<T>`:

```
public class MyGenCollection : IEnumerable<int>
{
```

```
public class MyGenCollection : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };
  public IEnumerator<int> GetEnumerator()
  {
    foreach (int i in data)
      yield return i;
  }
```



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

  // Explicit implementation
  // keeps it hidden.
```

Because `IEnumerable<T>` implements `IEnumerable`, we must implement both the generic and the nongeneric versions of `GetEnumerator`. In accordance with standard

practice, we've implemented the nongeneric version explicitly. It can simply call the generic `GetEnumerator` because `IEnumerator<T>` implements `IEnumerator`.

The class we've just written would be suitable as a basis from which to write a more sophisticated collection. However, if we need nothing above a simple `IEnumerable<T>` implementation, the `yield return` statement allows for an easier variation. Rather than writing a class, you can move the iteration logic into a method returning a generic `IEnumerable<T>` and let the compiler take care of the rest. Here's an example:

```
public class Test
{
  public static IEnumerable <int> GetSomeIntegers()
  {
    yield return 1;
    yield return 2;
    yield return 3;
```

```
    yield return 3;
  }
}
```

Here's our method in use:

```
foreach (int i in Test.GetSomeIntegers())
  Console.WriteLine (i);
// Output
1
2
3
```

The final approach in writing `GetEnumerator` is to write a class that implements `IEnumerator` directly. This is exactly what the compiler does behind the scenes, in resolving iterators. (Fortunately, it's rare that you'll need to go this far yourself.) The following example defines a collection that's hardcoded to contain the integers 1, 2,

resolving iterators. (Fortunately, it's rare that you'll need to go this far yourself.) The following example defines a collection that's hardcoded to contain the integers 1, 2, and 3:

```
public class MyIntList : IEnumerable
{
  int[] data = { 1, 2, 3 };
  public IEnumerator GetEnumerator()
  {
    return new Enumerator (this);
  }
  class Enumerator : IEnumerator
  {
    MyIntList collection;
```

```
    MyIntList collection;
    int currentIndex = -1;

// Define an inner class
// for the enumerator.

internal Enumerator (MyIntList collection)
{
  this.collection = collection;
}
public object Current
```



```
{
  get
```

```
{
  get
  {
    if (currentIndex == -1)
      throw new InvalidOperationException ("Enumeration not started!");
    if (currentIndex == collection.data.Length)
      throw new InvalidOperationException ("Past end of list!");
    return collection.data [currentIndex];
  }
}
```

```
    public bool MoveNext()
    {
      if (currentIndex > collection.data.Length) return false;
      return ++currentIndex < collection.data.Length;
    }
    public void Reset() { currentIndex = -1; }
  }
}
```

Implementing `Reset` is optional—you can instead throw a

Implementing `Reset` is optional—you can instead throw a `NotSupportedException`.

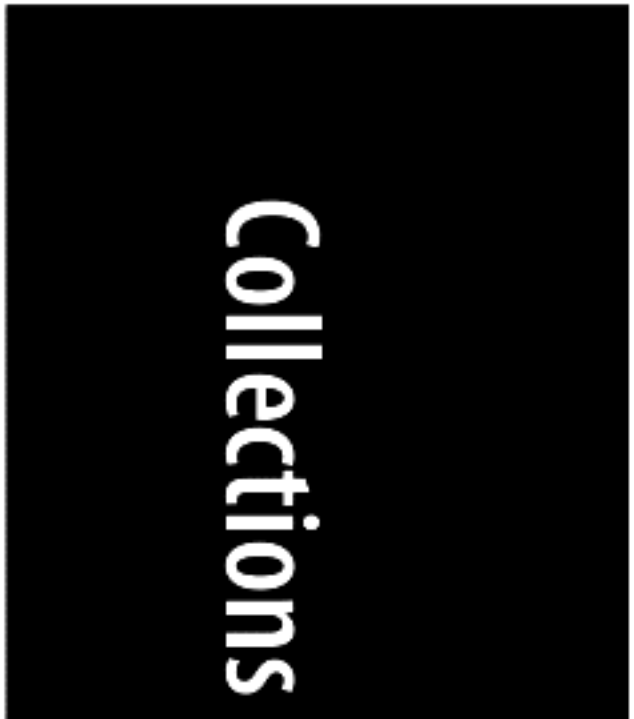
Note that the first call to `MoveNext` should move to the first (and not the second) item in the list.

To get on par with an iterator in functionality, we must also implement `IEnumerator<T>`. Here's an example with bounds checking omitted for brevity:

```
class MyIntList : IEnumerable<int>
{
  int[] data = { 1, 2, 3 };

// The generic enumerator is compatible with both IEnumerable and
// IEnumerable<T>. We implement the nongeneric GetEnumerator method
// explicitly to avoid a naming conflict.

public IEnumerator<int> GetEnumerator() { return new Enumerator(this); }
```

```
public IEnumerator<int> GetEnumerator() { return new Enumerator(this); }
IEnumerator IEnumerable.GetEnumerator() { return new Enumerator(this); }
```

```
class Enumerator : IEnumerator<int>
{
  int currentIndex = -1;
  MyIntList collection;
```



```
internal Enumerator (MyIntList collection)
{
  this.collection = collection;
}
```

```
public int Current { get { return collection.data [currentIndex]; } }
object IEnumerator.Current { get { return Current; } }
```



```
public bool MoveNext()
{
  return ++currentIndex < collection.data.Length;
}
```

```
    return ++currentIndex < collection.data.Length;
  }
  public void Reset() { currentIndex = -1; }

  // Given we don't need a Dispose method, it's good practice to
  // implement it explicitly, so it's hidden from the public interface.

      void IDisposable.Dispose() {}
    }
  }
```

The example with generics is faster because `IEnumerator<int>.Current` doesn't require casting from `int` to `object`, and so avoids the overhead of boxing.

When to Use the Nongeneric Interfaces

Given the extra type safety of the generic collection interfaces such as `IEnumerable<T>`, the question arises: do you ever need to use the nongeneric

Given the extra type safety of the generic collection interfaces such as `IEnumerable<T>`, the question arises: do you ever need to use the nongeneric `IEnumerable` (or `ICollection` or `IList`)?

In the case of `IEnumerable`, you must implement this interface in conjunction with `IEnumerable<T>`—because the latter derives from the former. However, it's very rare that you actually implement these interfaces from scratch: in nearly all cases, you can take the higher-level approach of using iterator methods, `Collection<T>`, and LINQ.

So, what about as a consumer? In nearly all cases, you can manage entirely with the generic interfaces. The nongeneric interfaces are still occasionally useful, though, in their ability to provide type unification for collections across all element types. The following method, for instance, counts elements in any collection *recursively*:

```
public static int Count (IEnumerable e)
{
  int count = 0;
  foreach (object element in e)
  {
    var subCollection = element as IEnumerable;
    if (subCollection != null)
      count += Count (subCollection);
    else
      count++;
  }
```

```
            count++;
        }
        return count;
    }
```

Because C# 4.0 offers covariance with generic interfaces, it might seem valid to have this method instead accept `IEnumerable<object>`. This would fail with value-type elements, however. It would also fail with legacy collections that don't implement `IEnumerable<T>`— an example is `ControlCollection` in Windows Forms.

The ICollection and IList Interfaces

Although the enumeration interfaces provide a protocol for forward-only iteration over a collection, they don't provide a mechanism to determine the size of the collection, access a member by index, search, or modify the collection. For such functionality, the .NET Framework defines the `ICollection`, `IList`, and `IDictionary` interfaces. Each comes in both generic and nongeneric versions; however, the nongeneric versions exist mostly for legacy.

The inheritance hierarchy for these interfaces was shown in Figure 7-1. The easiest

generic versions exist mostly for legacy).

The inheritance hierarchy for these interfaces was shown in Figure 7-1. The easiest way to summarize them is as follows:

`IEnumerable<T>` *(and* `IEnumerable`*)*
: Provides minimum functionality (enumeration only)

`ICollection<T>` *(and* `ICollection`*)*
: Provides medium functionality (e.g., the `Count` property)

`IList <T>/IDictionary <K,V>` *and their nongeneric versions*
: Provide maximum functionality (including "random" access by index/key)

It's rare that you'll need to *implement* any of these interfaces. In nearly all cases when you need to write a collection class, you can instead subclass `Collection<T>` (see "Customizable Collections and Proxies" on page 298). LINQ provides yet another option that covers many scenarios.

The generic and nongeneric versions differ in ways over and above what you might expect, particularly in the case of `ICollection`. The reasons for this are mostly historical: because generics came later, the generic interfaces were developed with the

torical: because generics came later, the generic interfaces were developed with the benefit of hindsight. For this reason, `ICollection<T>` does not extend `ICollection`, `IList<T>` does not extend `IList`, and `IDictionary<TKey, TValue>` does not extend `IDictionary`. Of course, a collection class itself is free to implement both versions of an interface if beneficial (which, often, it is).

Another, subtler reason for `IList<T>` not extending `IList` is that casting to `IList<T>` would then return an interface with both `Add(T)` and `Add(object)` members. This would effectively defeat static type safety, because you could call `Add` with an object of any type.

This section covers `ICollection<T>`, `IList<T>`, and their nongeneric versions; "Dictionaries" on page 292 covers the dictionary interfaces.

There is no *consistent* rationale in the way the words *collection* and *list* are applied throughout the .NET Framework. For instance, since `IList<T>` is a more functional version of `ICollection<T>`, you might expect the class `List<T>` to be correspondingly more functional than the class `Collection<T>`. This is not the case. It's best to consider the terms *collection* and *list* as broadly synonymous, except when a specific type is involved.

involved.

ICollection<T> and ICollection

`ICollection<T>` is the standard interface for countable collections of objects. It provides the ability to determine the size of a collection (`Count`), determine whether an item exists in the collection (`Contains`), copy the collection into an array (`ToArray`), and determine whether the collection is read-only (`IsReadOnly`). For writable collections, you can also `Add`, `Remove`, and `Clear` items from the collection. And since it extends `IEnumerable<T>`, it can also be traversed via the `foreach` statement:

```
public interface ICollection<T> : IEnumerable<T>, IEnumerable
{
  int Count { get; }
```

```
 bool Contains (T item);
 void CopyTo (T[] array, int arrayIndex);
 bool IsReadOnly { get; }
```

```
    void Add(T item);
    bool Remove (T item);
    void Clear();
}
```

The nongeneric `ICollection` is similar in providing a countable collection, but doesn't provide functionality for altering the list or checking for element membership:

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
```

```
    object SyncRoot { get; }
    void CopyTo (Array array, int index);
}
```

The nongeneric interface also defines properties to assist with synchronization (Chapter 21)—these were dumped in the generic version because thread safety is no longer considered intrinsic to the collection.

Both interfaces are fairly straightforward to implement. If implementing a read-only `ICollection<T>`, the `Add`, `Remove`, and `Clear` methods should throw a `NotSupportedException`.

These interfaces are usually implemented in conjunction with either the `IList` or the `IDictionary` interface.

IList<T> and IList

`IList<T>` is the standard interface for collections indexable by position. In addition

`IList<T>` is the standard interface for collections indexable by position. In addition to the functionality inherited from `ICollection<T>` and `IEnumerable<T>`, it provides the ability to read or write an element by position (via an indexer) and insert/remove by position:

```
public interface IList<T> : ICollection<T>, IEnumerable<T>, IEnumerable
{
  T this [int index] { get; set; }
  int IndexOf (T item);
  void Insert (int index, T item);
  void RemoveAt (int index);
}
```

The `IndexOf` methods perform a linear search on the list, returning `-1` if the specified item is not found.

The nongeneric version of `IList` has more members because it inherits less from `ICollection`:

```
public interface IList : ICollection, IEnumerable
{
```

```
{
  object this [int index] { get; set }
  bool IsFixedSize { get; }
  bool IsReadOnly  { get; }
  int  Add      (object value);
  void Clear();
  bool Contains (object value);
  int  IndexOf  (object value);
  void Insert   (int index, object value);
  void Remove   (object value);
  void RemoveAt (int index);
}
```

–

The `Add` method on the nongeneric `IList` interface returns an integer—this is the index of the newly added item. In contrast, the `Add` method on `ICollection<T>` has a `void` return type.

–

The general-purpose `List<T>` class is the quintessential implementation of both

The general-purpose `List<T>` class is the quintessential implementation of both `IList<T>` and `IList`. C# arrays also implement both the generic and nongeneric `ILists` (although the methods that add or remove elements are hidden via explicit interface implementation and throw a `NotSupportedException` if called).

The Array Class

The `Array` class is the implicit base class for all single and multidimensional arrays, and it is one of the most fundamental types implementing the standard collection interfaces. The `Array` class provides type unification, so a common set of methods is available to all arrays, regardless of their declaration or underlying element type.

Since arrays are so fundamental, C# provides explicit syntax for their declaration and initialization, described in Chapters 2 and 3. When an array is declared using C#'s syntax, the CLR implicitly subtypes the `Array` class—synthesizing a

pseudotype appropriate to the array's dimensions and element types. This pseudotype implements the typed generic collection interfaces, such as `IList<string>`.

The CLR also treats array types specially upon construction, assigning them a contiguous space in memory. This makes indexing into arrays highly efficient, but prevents them from being resized later on.

`Array` implements the collection interfaces up to `IList<T>` in both their generic and nongeneric forms. `IList<T>` itself is implemented explicitly, though, to keep `Array`'s public interface clean of methods such as `Add` or `Remove`, which throw an

nongeneric forms. `IList<T>` itself is implemented explicitly, though, to keep `Array`'s public interface clean of methods such as `Add` or `Remove`, which throw an exception on fixed-length collections such as arrays. The `Array` class does actually offer a static `Resize` method, although this works by creating a new array and then copying over each element. As well as being inefficient, references to the array elsewhere in the program will still point to the original version. A better solution for resizable collections is to use the `List<T>` class (described in the following section).

An array can contain value type or reference type elements. Value type elements are stored in place in the array, so an array of three long integers (each 8 bytes) will occupy 24 bytes of contiguous memory. A reference type element, however, occupies only as much space in the array as a reference (4 bytes in a 32-bit environment or 8 bytes in a 64-bit environment). Figure 7-2 illustrates the effect, in memory, of the following program:

```
StringBuilder[] builders = new StringBuilder [5];
builders [0] = new StringBuilder ("builder1");
builders [1] = new StringBuilder ("builder2");
builders [2] = new StringBuilder ("builder3");
```

```
long[] numbers = new long [3];
numbers [0] = 12345;
```

```
numbers [0] = 12345;
numbers [1] = 54321;
```

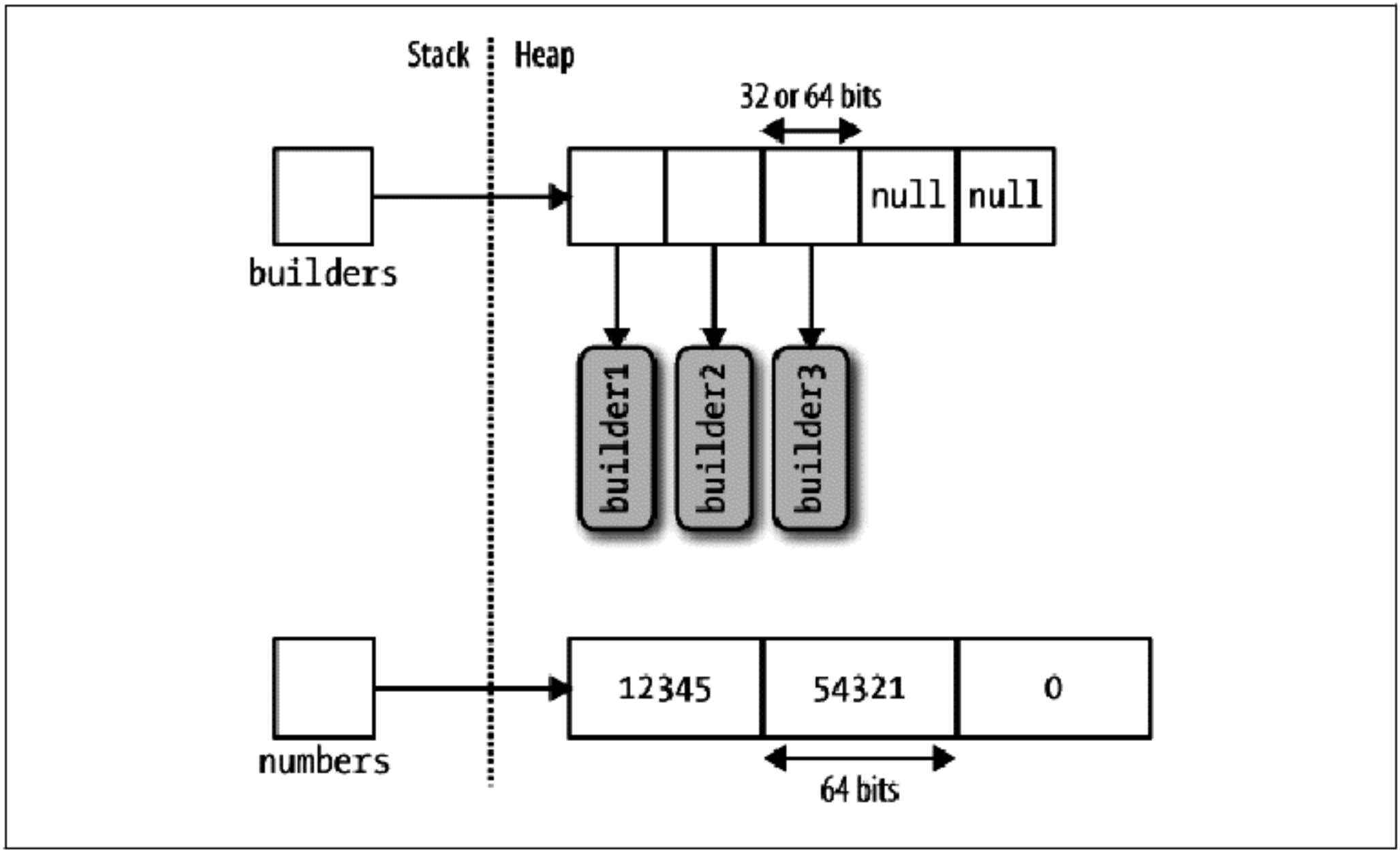


Figure 7-2. Arrays in memory

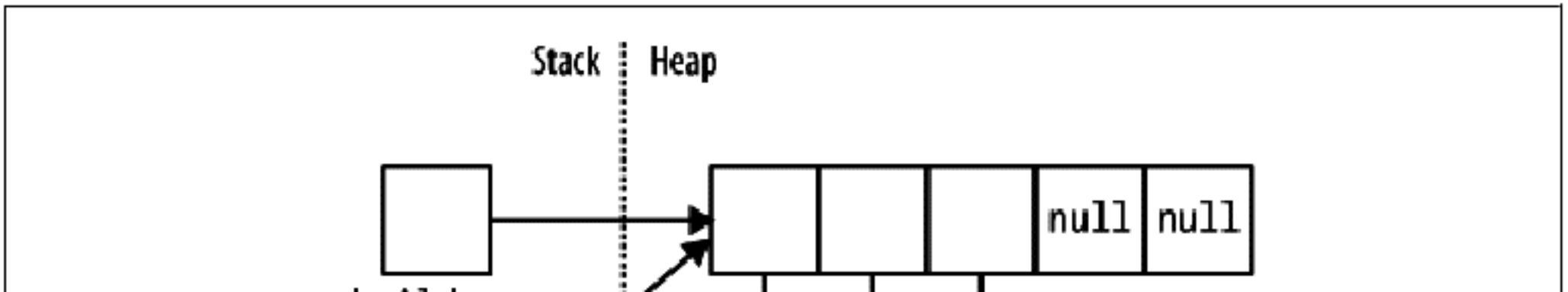
Because `Array` is a class, arrays are always (themselves) reference types—regardless of the array's element type. This means that the statement `arrayB = arrayA` results in two variables that reference the same array. Similarly, two distinct arrays will always fail an equality test—unless you use a custom equality comparer. Framework 4.0 provides one for the purpose of comparing elements in arrays or tuples which you can access via the `StructuralComparisons` type:

```
object[] a1 = { "string", 123, true };
object[] a2 = { "string", 123, true };

Console.WriteLine (a1 == a2);
Console.WriteLine (a1.Equals (a2));

Console.WriteLine (a1.Equals (a2,
 StructuralComparisons.StructuralEqualityComparer));

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

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

Arrays can be duplicated with the `Clone` method: `arrayB = arrayA.Clone()`. However, this results in a shallow clone, meaning that only the memory represented by the array itself is copied. If the array contains value type objects, the values themselves are copied; if the array contains reference type objects, just the references are copied (resulting in two arrays whose members reference the same objects). Figure 7-3 demonstrates the effect of adding the following code to our example:

```
StringBuilder[] builders2 = builders;
StringBuilder[] shallowClone = (StringBuilder[]) builders.Clone();
```



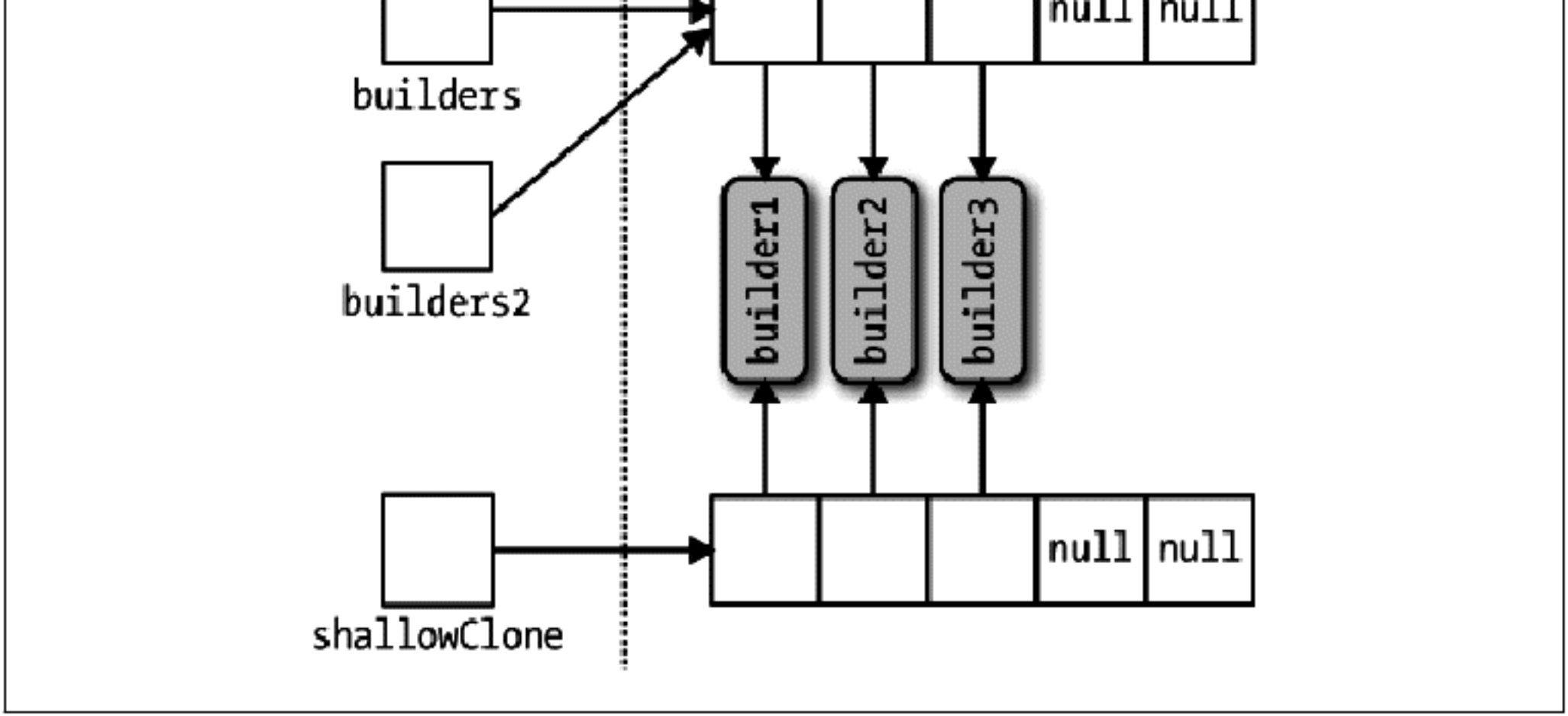
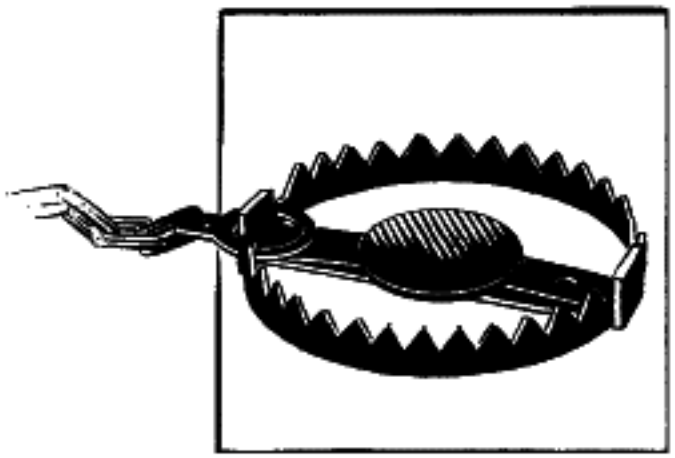


Figure 7-3. Shallow-cloning an array

To create a deep copy—where reference type subobjects are duplicated—you must loop through the array and clone each element manually. The same rules apply to other .NET collection types.

Although `Array` is designed primarily for use with 32-bit indexers, it also has limited support for 64-bit indexers (allowing an array to theoretically address up to 2^{64}

elements) via several methods that accept both `Int32` and `Int64` parameters. These overloads are useless in practice, because the CLR does not permit any object—including arrays—to exceed 2GB in size (whether running on a 32- or 64-bit environment).



Many of the methods on the `Array` class that you expect to be instance methods are in fact static methods. This is an odd design decision, and means you should check for both static and instance methods when looking for a method on `Array`.

Construction and Indexing

The easiest way to create and index arrays is through C#'s language constructs:

```
int[] myArray = { 1, 2, 3 };
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

```
int first = myArray [0];
int last = myArray [myArray.Length - 1];
```

Alternatively, you can instantiate an array dynamically by calling `Array.CreateInstance`. This allows you to specify element type and rank (number of dimensions) at runtime, as well as allowing nonzero-based arrays through specifying a lower bound. Nonzero-based arrays are not CLS (Common Language Specification)-compliant.

The static `GetValue` and `SetValue` methods let you access elements in a dynamically created array (they also work on ordinary arrays):

```
// Create a string array 2 elements in length:
Array a = Array.CreateInstance (typeof(string), 2);
a.SetValue ("hi", 0);                          //  → a[0] = "hi";
a.SetValue ("there", 1);                       //  → a[1] = "there";
string s = (string) a.GetValue (0);            //  → s = a[0];
```

```
// We can also cast to a C# array as follows:
string[] cSharpArray = (string[]) a;
string s2 = cSharpArray [0];
```

Zero-indexed arrays created dynamically can be cast to a C# array of a matching or compatible type (compatible by standard array-variance rules). For example, if `Apple` subclasses `Fruit`, `Apple[]` can be cast to `Fruit[]`. This leads to the issue of why `object[]` was not used as the unifying array type rather the `Array` class. The answer is that `object[]` is incompatible with both multidimensional and value-type arrays (and nonzero-based arrays). An `int[]` array cannot be cast to `object[]`. Hence, we require the `Array` class for full type unification.

`GetValue` and `SetValue` also work on compiler-created arrays, and they are useful when writing methods that can deal with an array of any type and rank. For multidimensional arrays, they accept an *array* of indexers:

```
public object GetValue (params int[] indices)
public void   SetValue (object value, params int[] indices)
```

The following method prints the first element of any array, regardless of rank:


```
void WriteFirstValue (Array a)
{
```

```
void WriteFirstValue (Array a)
{
  Console.Write (a.Rank + "-dimensional; ");

// The indexers array will automatically initialize to all zeros, so
// passing it into GetValue or SetValue will get/set the zero-based
// (i.e., first) element in the array.

  int[] indexers = new int[a.Rank];
  Console.WriteLine ("First value is " +
}
void Demo()
{
  int[]  oneD = { 1, 2, 3 };
  int[,] twoD = { {5,6}, {8,9} };

a.GetValue (indexers));
```

```
a.GetValue (indexers));

  WriteFirstValue (oneD);
  WriteFirstValue (twoD);
}

// 1-dimensional; first value is 1
// 2-dimensional; first value is 5
```

For working with arrays of unknown type but known rank, generics provide an easier and more efficient solution:

```
void WriteFirstValue<T> (T[] array)
{
  Console.WriteLine (array[0]);
}
```

```
    Console.WriteLine (array[0]);
}
```

`SetValue` throws an exception if the element is of an incompatible type for the array.

When an array is instantiated, whether via language syntax or `Array.CreateInstance`, its elements are automatically initialized. For arrays with reference type elements, this means writing nulls; for arrays with value type elements, this means calling the value type's default constructor (effectively "zeroing" the members). The `Array` class also provides this functionality on demand via the `Clear` method:

```
public static void Clear (Array array, int index, int length);
```

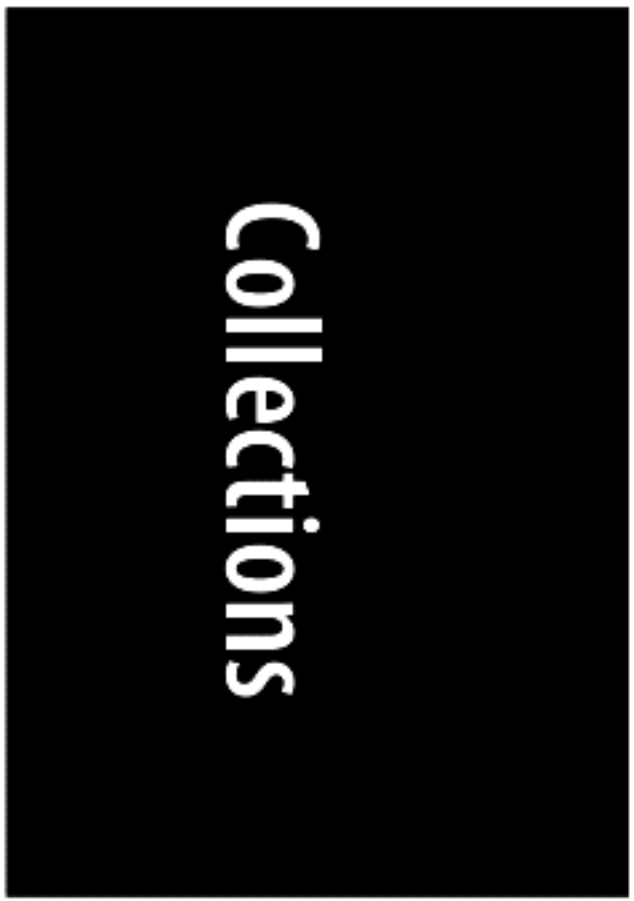
This method doesn't affect the size of the array. This is in contrast to the usual use of `Clear` (such as in `ICollection<T>.Clear`), where the collection is reduced to zero elements.

Enumeration

Arrays are easily enumerated with a `foreach` statement:

```
int[] myArray = { 1, 2, 3};
foreach (int val in myArray)
```

```
int[] myArray = { 1, 2, 3 };
foreach (int val in myArray)
  Console.WriteLine (val);
```



You can also enumerate using the static `Array.ForEach` method, defined as follows:

```
public static void ForEach<T> (T[] array, Action<T> action);
```

This uses an `Action` delegate, with this signature:

```
public delegate void Action<T> (T obj);
```

Here's the first example rewritten with `Array.ForEach`:

```
Array.ForEach (new[] { 1, 2, 3 }, Console.WriteLine);
```

Length and Rank

`Array` provides the following methods and properties for querying length and rank:

```
public int  GetLength     (int dimension);
public long GetLongLength (int dimension);

public int  Length        { get; }
public long LongLength    { get; }
```

```
public long LongLength   { get; }
public int GetLowerBound (int dimension);
public int GetUpperBound (int dimension);

public int Rank { get; }

// Returns number of dimensions in array
```

`GetLength` and `GetLongLength` return the length for a given dimension (`0` for a single-dimensional array), and `Length` and `LongLength` return the total number of elements in the array—all dimensions included.

`GetLowerBound` and `GetUpperBound` are useful with nonzero indexed arrays. `GetUpperBound` returns the same result as adding `GetLowerBound` to `GetLength` for any given dimension.

Searching

Searching

The `Array` class provides a range of methods for finding elements within a one-dimensional array:

```
public static int BinarySearch<T> (T[] array,   object value);
public static int BinarySearch<T> (T[] array,   object value, IComparer<T>
                                                comparer);
public static int BinarySearch    (Array array, object value);
public static int BinarySearch    (Array array, object value, IComparer
                                                comparer);
public static int IndexOf<T>      (T[] array,   T value);
public static int IndexOf         (Array array, object value);
public static int LastIndexOf<T>  (T[] array,   T value);
public static int LastIndexOf     (Array array, object value);
```

```
// Predicate-based searching:
public static T    Find<T>
public static T    FindLast<T>
public static T[]  FindAll<T>
```

```
public static T[]  FindAll<T>
public static bool Exists<T>
public static bool TrueForAll<T>

(T[] array, Predicate<T> match);
(T[] array, Predicate<T> match);
(T[] array, Predicate<T> match);

(T[] array, Predicate<T> match);
(T[] array, Predicate<T> match);
```



```
public static int
```

```
public static int
public static int
```

```
FindIndex<T>     (T[] array, Predicate<T> match);
FindLastIndex<T> (T[] array, Predicate<T> match);
```

The methods shown in bold are also overloaded to accept the following additional arguments:

```
int index   // starting index at which to begin searching
int length  // maximum number of elements to search
```

None of these methods throws an exception if the specified value is not found. Instead, if an item is not found, methods returning an integer return `-1` (assuming a zero-indexed array), and methods returning a generic type return the type's default value (e.g., `0` for an `integer`, or `null` for a `string`).

The binary search methods are fast, but they work only on sorted arrays and require that the elements be compared for *order*, rather than simply *equality*. To this effect, the binary search methods can accept an `IComparer` or `IComparer<T>` object to arbitrate on ordering decisions (see the section "Plugging in Equality and Or-

the binary search methods can accept an `IComparer` or `IComparer<T>` object to arbitrate on ordering decisions (see the section "Plugging in Equality and Order" on page 304, later in this chapter). This must be consistent with any comparer used in originally sorting the array. If no comparer is provided, the type's default ordering algorithm will be applied, based on its implementation of `IComparable`/`IComparable<T>`.

The `IndexOf` and `LastIndexOf` methods perform a simple enumeration over the array, returning the position of the first (or last) element that matches the given value.

The predicate-based searching methods allow a method delegate or lambda expression to arbitrate on whether a given element is a "match." A predicate is simply a delegate accepting an object and returning `true` or `false`:

```
public delegate bool Predicate<T> (T object);
```

In the following example, we search an array of strings for a name containing the letter "a":

```
static void Main()
{
  string[] names = { "Rodney", "Jack", "Jill" };
  string match = Array.Find (names, ContainsA);
```

```
  string[] names = { "Rodney", "Jack", "Jill" };
  string match = Array.Find (names, ContainsA);
  Console.WriteLine (match);     // Jack
}
static bool ContainsA (string name) { return name.Contains ("a"); }
```

Here's the same code shortened with an anonymous method:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, delegate (string name)
  { return name.Contains ("a"); } );
```

A lambda expression shortens it further:

```
string[] names = { "Rodney", "Jack", "Jill" };
string match = Array.Find (names, n => n.Contains ("a"));
```

```
// Jack
```

`FindAll` returns an array of all items satisfying the predicate. In fact, it's equivalent to `Enumerable.Where` in the `System.Linq` namespace, except that `FindAll` returns an array of matching items rather than an `IEnumerable<T>` of the same.

`Exists` returns `true` if any array member satisfies the given predicate, and is equivalent to `Any` in `System.Linq.Enumerable`.

`TrueForAll` returns `true` if all items satisfy the predicate, and is equivalent to `All` in

`TrueForAll` returns `true` if all items satisfy the predicate, and is equivalent to `All` in `System.Linq.Enumerable`.

Sorting

`Array` has the following built-in sorting methods:

```
// For sorting a single array:
public static void Sort<T> (T[] array);
public static void Sort    (Array array);
// For sorting a pair of arrays:
public static void Sort<TKey,TValue> (TKey[] keys, TValue[] items);
public static void Sort              (Array keys, Array items);
```

Each of these methods is additionally overloaded to also accept:

```
int index                  // Starting index at which to begin sorting
int length                 // Number of elements to sort
IComparer<T> comparer      // Object making ordering decisions
```

```
int length                    // Number of elements to sort
IComparer<T> comparer         // Object making ordering decisions
Comparison<T> comparison      // Delegate making ordering decisions
```

The following illustrates the simplest use of `Sort`:

```
int[] numbers = { 3, 2, 1 };
Array.Sort (numbers);                          // Array is now { 1, 2, 3 }
```

The methods accepting a pair of arrays work by rearranging the items of each array in tandem, basing the ordering decisions on the first array. In the next example, both the numbers and their corresponding words are sorted into numerical order:

```
int[] numbers = { 3, 2, 1 };
string[] words = { "three", "two", "one" };
Array.Sort (numbers, words);

// numbers array is now { 1, 2, 3 }
// words   array is now { "one", "two", "three" }
```

`Array.Sort` requires that the elements in the array implement `IComparable` (see the section "Order Comparison" on page 255 in Chapter 6). This means that most primitive C# types (such as integers, as in the preceding example) can be sorted. If

primitive C# types (such as integers, as in the preceding example) can be sorted. If the elements are not intrinsically comparable, or you want to override the default ordering, you must provide `Sort` with a custom `comparison` provider that reports on the relative position of two elements. There are ways to do this:

-
-

Via a helper object that implements `IComparer` /`IComparer<T>` (see the section "Plugging in Equality and Order" on page 304)

Via a `Comparison` delegate:

```
public delegate int Comparison<T> (T x, T y);
```


The `Comparison` delegate follows the same semantics as `IComparer<T>.CompareTo`: if `x` comes before `y`, a negative integer is returned; if `x` comes after `y`, a positive integer is returned; if `x` and `y` have the same sorting position, `0` is returned.

In this example, we sort an array of integers so that the odd numbers come first:

In this example, we sort an array of integers so that the odd numbers come first:

```
int[] numbers = { 1, 2, 3, 4, 5 };
Array.Sort (numbers, (x, y) => x % 2 == y % 2 ? 0 : x % 2 == 1 ? -1 : 1);

// numbers array is now { 3, 5, 1, 2, 4 }
```

As an alternative to calling `Sort`, you can use LINQ's `OrderBy` and `ThenBy` operators. Unlike `Array.Sort`, the LINQ operators don't alter the original array, instead emitting the sorted result in a fresh `IEnumerable<T>` sequence.

Reversing Elements

These `Array` methods reverse the order of all—or a part of—elements in an array:

```
public static void Reverse (Array array);
public static void Reverse (Array array, int index, int length);
```

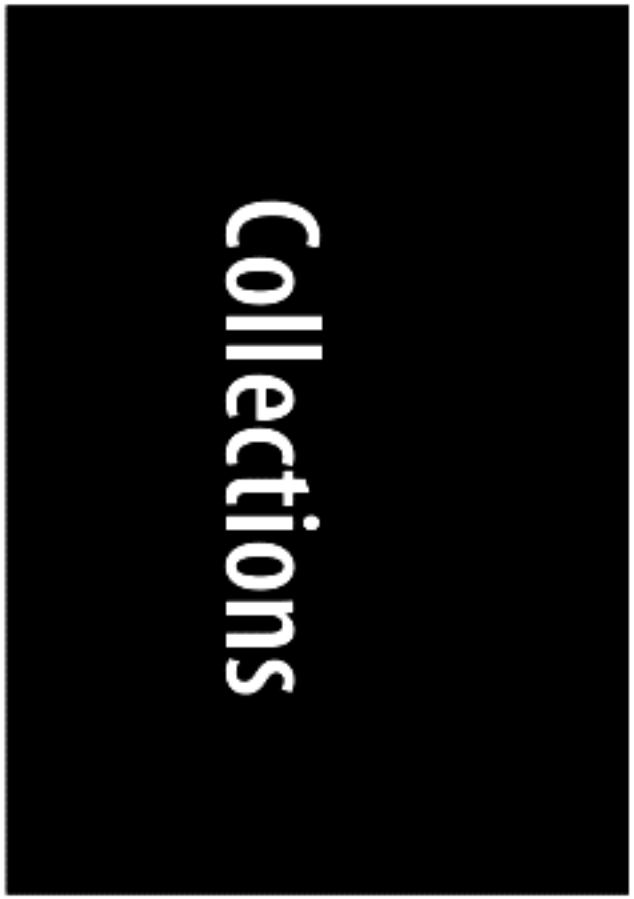
Copying, Converting, and Resizing

Copying, Converting, and Resizing

`Array` provides shallow copying and cloning methods as follows:

```
    // Instance methods:

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

// Static methods:

public static void Copy (Array sourceArray,
                         Array destinationArray,
                         int length);

public static void Copy (Array sourceArray,      int sourceIndex,
                         Array destinationArray, int destinationIndex,
                         int length);
```

```
public static void ConstrainedCopy (
                        Array sourceArray,      int sourceIndex,
                        Array destinationArray, int destinationIndex,
                        int length);
```




```
public static ReadOnlyCollection<T> AsReadOnly<T> (T[] array)
```

```
public static TOutput[] ConvertAll<TInput, TOutput>
  (TInput[] array, Converter<TInput, TOutput> converter)

public static void Resize<T> (ref T[] array, int newSize);
```


The `Copy` and `CopyTo` methods are overloaded to accept `Int64` index arguments.

The `Clone` method returns a whole new (shallow-copied) array. The `Copy` and `CopyTo` methods copy a contiguous subset of the array. Copying a multidimensional rectangular array requires you to map the multidimensional index to a linear index. For example, the middle square (`position[1,1]`) in a 3 × 3 array is represented with the index 4, from the calculation: 1*3 + 1. The source and destination ranges can overlap without causing a problem.

`ConstrainedCopy` performs an *atomic* operation: if all of the requested elements cannot be successfully copied (due to a type error, for instance), the operation is rolled back.

back.

`AsReadOnly` returns a wrapper that prevents elements from being reassigned. `ConvertAll` creates and returns a new array of element type `TOutput`, calling the supplied `Converter` delegate to copy over the elements. `Converter` is defined as follows:

```
public delegate TOutput Converter<TInput,TOutput> (TInput input)
```

The following converts an array of floats to an array of integers:

```
float[] reals = { 1.3f, 1.5f, 1.8f };
int[] wholes = Array.ConvertAll (reals, r => Convert.ToInt32 (r));

// wholes array is { 1, 2, 2 }
```

The `Resize` method works by creating a new array and copying over the elements, returning the new array via the reference parameter. However, any references to the original array in other objects will remain unchanged.

The `System.Linq` namespace offers an additional buffet of extension methods suitable for array conversion. These methods return an `IEnumerable<T>`, which you can convert back to an

tension methods suitable for array conversion. These methods return an `IEnumerable<T>`, which you can convert back to an array via `Enumerable`'s `ToArray` method.

Lists, Queues, Stacks, and Sets

The Framework provides a comprehensive set of concrete collection classes that implement the interfaces described in this chapter. This section concentrates on the *list-like* collections (versus the *dictionary-like* collections covered in "Dictionaries" on page 292). As with the interfaces we discussed previously, you usually have a choice of generic or nongeneric versions of each type. In terms of flexibility and performance, the generic classes win, making their nongeneric counterparts redundant except for backward compatibility. This differs from the situation with collection interfaces, where the nongeneric versions are still occasionally useful.

Of the classes described in this section, the generic `List` class is the most commonly used.

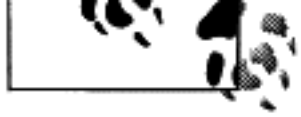
List<T> and ArrayList

List<T> and ArrayList

The generic `List` and nongeneric `ArrayList` classes provide a dynamically sized array of objects and are among the most commonly used of the collection classes. `ArrayList` implements `IList`, whereas `List<T>` implements both `IList` and `IList<T>`. Unlike with arrays, all interfaces are implemented publicly, and methods such as `Add` and `Remove` are exposed and work as you would expect.

Internally, `List<T>` and `ArrayList` work by maintaining an internal array of objects, replaced with a larger array upon reaching capacity. Appending elements is efficient (since there is usually a free slot at the end), but inserting elements can be slow (since all elements after the insertion point have to be shifted to make a free slot). As with arrays, searching is efficient if the `BinarySearch` method is used on a list that has been sorted, but is otherwise inefficient because each item must be individually checked.

`List<T>` is up to several times faster than `ArrayList` if `T` is a value type, because `List<T>` avoids the overhead of boxing and unboxing elements.



boxing elements.

List<T> and ArrayList provide constructors that accept an existing collection of elements: these copy each element from the existing collecting into the new List<T> or ArrayList:

```
public class List <T> : IList <T>
{
   public List ();
   public List (IEnumerable<T> collection);
   public List (int capacity);
```

```
// Add+Insert
public void Add         (T item);
public void AddRange    (IEnumerable<T> collection);
public void Insert      (int index, T item);
public void InsertRange (int index, IEnumerable<T> collection);
```

```
// Remove
public bool Remove      (T item);
public void RemoveAt    (int index);
public void RemoveRange (int index, int count);
public int  RemoveAll   (Predicate<T> match);

// Indexing
public T this [int index] { get; set; }
public List<T> GetRange (int index, int count);
public Enumerator<T> GetEnumerator();
```


```
// Exporting, copying and converting:
public T[] ToArray();
public void CopyTo (T[] array);
public void CopyTo (T[] array, int arrayIndex);
public void CopyTo (int index, T[] array, int arrayIndex, int count);
```



```
  public ReadOnlyCollection<T> AsReadOnly();
  public List<TOutput> ConvertAll<TOutput> (Converter <T,TOutput>
                                            converter);
  // Other:
  public void Reverse();              // Reverses order of elements in list.
  public int Capacity { get;set; }  // Forces expansion of internal array.
```

```
  public void Reverse();                  // Reverses order of elements in list.
  public int Capacity { get;set; }  // Forces expansion of internal array.
  public void TrimExcess();               // Trims internal array back to size.
  public void Clear();                    // Removes all elements, so Count=0.
}
```

```
public delegate TOutput Converter <TInput, TOutput> (TInput input);
```

In addition to these members, `List<T>` provides instance versions of all of `Array`'s searching and sorting methods.

The following code demonstrates `List`'s properties and methods. See "The Array Class" on page 273 for examples on searching and sorting:

```
List<string> words = new List<string>();    // New string-typed list

words.Add ("melon");
words.Add ("avocado");
words.AddRange (new[] { "banana", "plum" } );
words.Insert (0, "lemon");
words.InsertRange (0, new[] { "peach", "nashi" });
```

```
words.InsertRange (0, new[] { "peach", "nashi" });

// Insert at start
// Insert at start

words.Remove ("melon");
words.RemoveAt (3);
words.RemoveRange (0, 2);

// Remove all strings starting in 'n':
words.RemoveAll (s => s.StartsWith ("n"));

// Remove the 4th element
// Remove first 2 elements

Console.WriteLine (words [0]);
```

```
Console.WriteLine (words [0]);
Console.WriteLine (words [words.Count - 1]);
foreach (string s in words) Console.WriteLine (s);
List<string> subset = words.GetRange (1, 2);

// first word
// last word
// all words
// 2nd->3rd words

string[] wordsArray = words.ToArray();

// Creates a new typed array

// Copy first two elements to the end of an existing array:
```

```
// Copy first two elements to the end of an existing array:
string[] existing = new string [1000];
words.CopyTo (0, existing, 998, 2);

List<string> upperCastWords = words.ConvertAll (s => s.ToUpper());
List<int> lengths = words.ConvertAll (s => s.Length);
```

The nongeneric `ArrayList` class is used mainly for backward compatibility with Framework 1.x code and requires clumsy casts—as the following example demonstrates:

```
ArrayList al = new ArrayList();
al.Add ("hello");
string first = (string) al [0];
string[] strArr = (string[]) al.ToArray (typeof (string));
```


Such casts cannot be verified by the compiler; the following compiles successfully but then fails at runtime:

but then fails at runtime:

```
int first = (int) al [0];

// Runtime exception
```

An `ArrayList` is functionally similar to `List<object>`. Both are useful when you need a list of mixed-type elements that share no common base type. A possible advantage of choosing an `ArrayList`, in this case, would be if you need to deal with the list using reflection (Chapter 18). Reflection is easier with a nongeneric `ArrayList` than a `List<object>`.

If you import the `System.Linq` namespace, you can convert an `ArrayList` to a generic `List` by calling `Cast` and then `ToList`:

```
ArrayList al = new ArrayList();
al.AddRange (new[] { 1, 5, 9 } );
List<int> list = al.Cast<int>().ToList();
```

```
var list = al.Cast<int>().ToList();
```

`Cast` and `ToList` are extension methods in the `System.Linq.Enumerable` class, supported from .NET Framework 3.5.

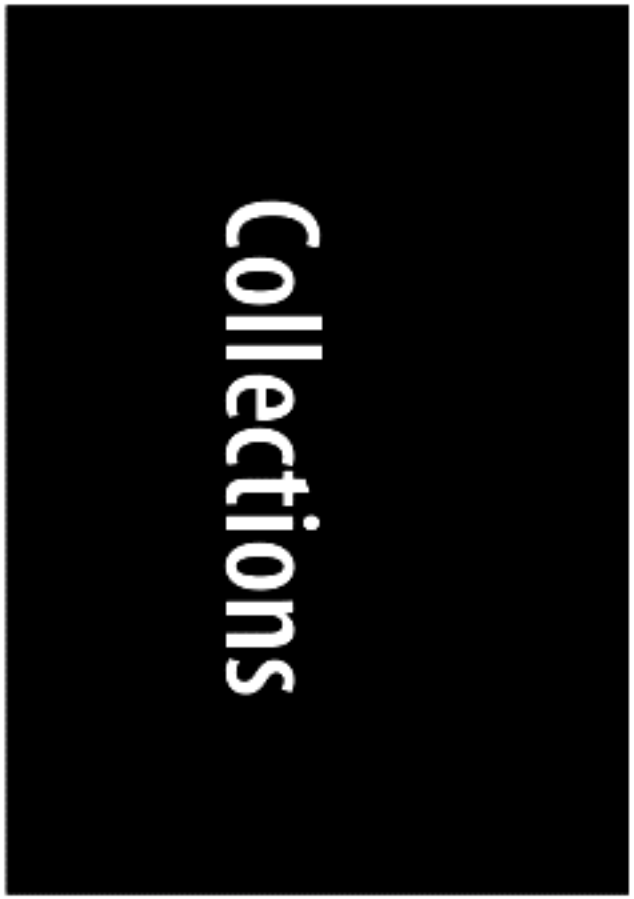
LinkedList<T>

`LinkedList<T>` is a generic doubly linked list (see Figure 7-4). A doubly linked list is a chain of nodes in which each references the node before, the node after, and the actual element. Its main benefit is that an element can always be inserted efficiently anywhere in the list, since it just involves creating a new node and updating a few references. However, finding where to insert the node in the first place can be slow as there's no intrinsic mechanism to index directly into a linked list; each node must be traversed, and binary-chop searches are not possible.

`LinkedList<T>` implements `IEnumerable<T>` and `ICollection<T>` (and their nongeneric versions), but not `IList<T>` since access by index is not supported. List nodes are implemented via the following class:

```
public sealed class LinkedListNode<T>
{
  public LinkedList<T> List { get; }
  public LinkedListNode<T> Next { get; }
```

```
    public LinkedList<T> List { get; }
    public LinkedListNode<T> Next { get; }
    public LinkedListNode<T> Previous { get; }
    public T Value { get; set; }
}
```



When adding a node, you can specify its position either relative to another node or at the start/end of the list. `LinkedList<T>` provides the following methods for this:

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