In addition to preconditions and postconditions, the code contracts API lets you make assertions and define *object invariants*.

Assertions

Contract.Assert

You can make assertions anywhere in a function by calling `Contract.Assert`. You can optionally specify an error message if the assertion fails:

```
...
int x = 3;
...
Contract.Assert (x == 3);
Contract.Assert (x == 3, "x must be 3");
// Fail unless x is 3
// Fail unless x is 3
...
```

The binary rewriter doesn’t move assertions around. There are two reasons for favoring `Contract.Assert` over `Debug.Assert`:

•

- You can leverage the more flexible failure-handling mechanisms offered by code contracts
- Static checking tools can attempt to validate `Contract.Asserts`

Contract.Assume

`Contract.Assume` behaves exactly like `Contract.Assert` at runtime, but has slightly different implications for static checking tools. Essentially, static checking tools won't *challenge* an assumption, whereas they may challenge an assertion. This is useful in that there will always be things a static checker is unable to prove, and this may lead to it "crying wolf" over a valid assertion. Changing the assertion to an assumption keeps the static checker quiet.

Object Invariants

For a class, you can specify one or more *object invariant* methods. These methods

For a class, you can specify one or more *object invariant* methods. These methods run automatically after every *public* function in the class, and allow you to assert that the object is in an internally consistent state.

Support for multiple object invariant methods was included to make object invariants work well with partial classes.

To define an object invariant method, write a parameterless void method and annotate it with the [ContractInvariantMethod] attribute. In that method, call Contract.Invariant to enforce each condition that should hold true if your object is a valid state:

```
class Test
{
  int x, y;
  int _x, _y;
```

```
[ContractInvariantMethod]
void ObjectInvariant()
{
  Contract.Invariant (_x >= 0);
  Contract.Invariant (_y >= _x);
}
```

```
  public int X { get { return _x; } set { _x = value; } }
  public void Test1() { _x = -3; }
  void Test2()        { _x = -3; }
}
```

The binary rewriter translates the `X` property, `Test1` method, and `Test2` method to something equivalent to this:

```
public void X { get { return _x; } set { _x = value; ObjectInvariant(); } }
public void Test1() { _x = -3; ObjectInvariant(); }
void Test2()        { _x = -3; }    // No change because it's private
```

Object invariants don't *prevent* an object from entering an invalid state: they merely *detect* when that condition has occurred.

`Contract.Invariant` is rather like `Contract.Assert`, except that it can appear only in a method marked with the [`ContractInvariantMethod`] attribute. And conversely, a contract invariant method can only contain calls to `Contract.Invariant`.

A subclass can introduce its own object invariant method, too, and this will be checked in addition to the base class's invariant method. The caveat, of course, is that the check will take place only after a public method is called.

Contracts on Interfaces and Abstract Methods

Contracts on Interfaces and Abstract Methods

A powerful feature of code contracts is that you can attach conditions to interface members and abstract methods. The binary rewriter then automatically weaves these conditions into the members' concrete implementations.

A special mechanism lets specify a separate contract class for interfaces and abstract methods, so that you can write method bodies to house the contract conditions. Here's how it works:

```
[ContractClass (typeof (ContractForITest))]
interface ITest
{
  int Process (string s);
}
```

```
[ContractClassFor (typeof (ITest))]
```

```
[ContractClassFor (typeof (ITest))]
sealed class ContractForITest : ITest
{
  int ITest.Process (string s)     // Must use explicit implementation.
  {
    Contract.Requires (s != null);
    return 0;                      // Dummy value to satisfy compiler.
  }
}
```

Notice that we had to return a value when implementing `ITest.Process` to satisfy the compiler. The code that returns 0 will not run, however. Instead, the binary rewriter extracts just the conditions from that method, and weaves them into the real implementations of `ITest.Process`. This means that the contract class is never actually instantiated (and any constructors that you write will not execute).

You can assign a temporary variable within the contract block to make it easier to reference other members of the interface. For instance, if our `ITest` interface also defined a `Message` property of type `string`, we could write the following in `ITest.Process`:

```
int ITest.Process (string s)
{
  ITest test = this;
  Contract.Requires (s != test.Message);
  ...
```

```
    ...
}
```

This is easier than:

```
Contract.Requires (s != ((ITest)this).Message);
```

(Simply using `this.Message` won't work because `Message` must be explicitly implemented.) The process of defining contract classes for abstract classes is exactly the same, except that the contract class should be marked `abstract` instead of `sealed`.

Dealing with Contract Failure

The binary rewriter lets you specify what happens when a contract condition fails, via the `/throwonfailure` switch (or the *Assert on Contract Failure* checkbox in Visual Studio's *Contracts* tab in *Project Properties*).

If you don't specify `/throwonfailure`—or check *Assert on Contract Failure*—a dialog appears upon contract failure, allowing you to abort, debug or ignore the error.

If you don't specify `/throwonfailure`—or check *Assert on Contract Failure*—a dialog appears upon contract failure, allowing you to abort, debug or ignore the error.

There are a couple of nuances to be aware of:

- If the CLR is hosted (i.e., in SQL Server or Exchange), the host's escalation policy is triggered instead of a dialog appearing.
- Otherwise, if the current process can't pop up a dialog box to the user, `Environment.FailFast` is called.

The dialog is useful in debug builds for a couple of reasons:

-
-

It makes it easy to diagnose and debug contract failures on the spot—without having to re-run the program. This works regardless of whether Visual Studio having to re-run the program. This works regardless of whether Visual Studio is configured to break on first-chance exceptions. And unlike with exceptions in general, contract failure almost certainly means a bug in your code.

It lets you know about contract failure—even if a caller higher up in the stack "swallows" exceptions as follows:

```
try
{
  // Call some method whose contract fails
}
catch { }
```

The code above is considered an antipattern in most scenarios because it *masks* failures, including conditions that the author never anticipated.

If you specify the `/throwonfailure` switch—or uncheck *Assert on Contract Failure*

If you specify the `/throwonfailure` switch—or uncheck *Assert on Contract Failure* in Visual Studio—a `ContractException` is thrown upon failure. This is desirable for:

•

•

Release builds—where you would let the exception bubble up the stack and be treated like any other unexpected exception (perhaps by having a top-level exception handler log the error or invite the user to report it).

Unit testing environments— where the process of logging errors is automated.

`ContractException` cannot appear in a `catch` block because this type is not public. The rationale is that there's no reason that you'd want to *specifically* catch a `ContractException`—you'd want to catch it only as part of a general exception backstop.

The ContractFailed Event

When a contract fails, the static `Contract.ContractFailed` event fires before any fur-

When a contract fails, the static Contract.ContractFailed event fires before any further action is taken. If you handle this event, you can query the event arguments object for details of the error. You can also call SetHandled to prevent a ContractException from being subsequently thrown (or a dialog appearing).

Handling this event is particularly useful when /throwonfailure is specified, because it lets you log *all* contract failures—even if code higher in the call stack swallows exceptions as we described just before. A great example is with automated unit testing:

```
Contract.ContractFailed += (sender, args) =>
{
  string failureMessage = args.FailureKind + ": " + args.Message;
  // Log failureMessage with unit testing framework:
  // ...
  args.SetUnwind();
};
```

This handler logs all contract failures, while allowing the normal ContractException (or contract failure dialog) to run its course after the event handler has finished. Notice that we also call SetUnwind: this neutralizes the effect of any calls to

Notice that we also call `SetUnwind`: this neutralizes the effect of any calls to `SetHandled` from other event subscribers. In other words, it ensures that a `ContractException` (or dialog) will always follow after all event handlers have run.

If you throw an exception from within this handler, any other event handlers will still execute. The exception that you threw then populates the `InnerException` property of the `ContractException` that's eventually thrown.

Exceptions Within Contract Conditions

If an exception is thrown within a contract condition itself, then that exception propagates like any other—regardless of whether `/throwonfailure` is specified. The following method throws a `NullReferenceException` if called with a null string:

```
string Test (string s)
{
  Contract.Requires (s.Length > 0);
  ...
}
```

This precondition is essentially faulty. It should instead be:

```
Contract.Requires (!string.IsNullOrEmpty (s));
```

Selectively Enforcing Contracts

The binary rewriter offers two switches that strip away some or all contract checking: `/publicsurface` and `/level`. You can control these from Visual Studio via the ing: `/publicsurface` and `/level`. You can control these from Visual Studio via the *Code Contracts* tab of *Project Properties*. The `/publicsurface` switch tells the rewriter to check contracts only on public members. The `/level` switch has the following options:

None (Level 0)
: Strips out *all* contract verification

ReleaseRequires (Level 1)
: Enables only calls to the generic version of `Contract.Requires<TException>`

Preconditions (Level 2)
: Enables all preconditions (Level 1 plus normal preconditions)

Pre and Post (Level 3)
: Enables Level2 checking plus postconditions

Full (Level 4)
: Enables Level 3 checking plus object invariants and assertions (i.e., everything)

You typically enable full contract checking in debug build configurations.

You typically enable full contract checking in debug build configurations.

Contracts in Release Builds

When it comes to making release builds, there are two general philosophies:

- Favor safety and enable full contract checking
- Favor performance and disable all contract checking

If you're building a library for public consumption, though, the second approach creates a problem. Imagine that you compile and distribute library L in release mode with contract checking disabled. A client then builds project C in *debug* mode that references library L. Assembly C can then call members of L incorrectly without contract violations! In this situation, you actually want to enforce the parts of L's contract that ensure correct usage of L—in other words, the *preconditions* in L's *public* members.

public members.

The simplest way to resolve this is to enable `/publicsurface` checking in L with a level of *Preconditions* or *ReleaseRequires*. This ensures that the essential preconditions are enforced for the benefit of consumers, while incurring the performance cost of only those preconditions.

In extreme cases, you might not want to pay even this small performance price—in which case, you can take the more elaborate approach of *call-site checking*.

Call-Site Checking

Call-site checking moves precondition validation from *called* methods into *calling* methods (call sites). This solves the problem just described—by enabling consumers of library L to perform L's precondition validation themselves in debug configurations.

To enable call-site checking, you must first build a separate *contracts reference assembly*—a supplementary assembly that contains just the preconditions for the referenced assembly. To do this, you can either use the *ccrefgen* command-line tool, or proceed in Visual Studio as follows:

referenced assembly. To do this, you can either use the *ccrefgen* command-line tool, or proceed in Visual Studio as follows:

1. In the release configuration of the *referenced library* (L), go to the *Code Contracts* tab of *Project Properties* and disable runtime contract checking while ticking “Build a Contract Reference Assembly”. This then generates a supplementary contracts reference assembly (with the suffix *.contracts.dll*).
2. In the *release* configuration of the *referencing* assemblies, disable all contract checking.

3. In the *debug* configuration of the *referencing* assemblies, tick “Call-site Requires Checking.”

The third step is equivalent to calling *ccrewrite* with the `/callsiterequires` switch. It reads the preconditions from the contracts reference assembly and weaves them into the calling sites in the referencing assembly.

Static Contract Checking

Static Contract Checking

Code contracts permit *static contract checking*, whereby a tool analyzes contract conditions to find potential bugs in your program before it's run. For example, statically checking the following code generates a warning:

```
static void Main()
{
  string message = null;
  WriteLine (message);     // Static checking tool will generate warning
}
```

```
static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

You can run Microsoft's static contracts tool either from the command line via *cccheck*, or by enabling static contract checking in Visual Studio's project properties dialog (the commercial version is supported with Visual Studio *Premium and Ultimate editions* only).

For static checking to work, you may need to add preconditions and postconditions to your methods. To give a simple example, the following will generate a warning:

to your methods. To give a simple example, the following will generate a warning:

```
static void WriteLine (string s, bool b)
{
  if (b)
    WriteLine (s);    // Warning: requires unproven
}
```

```
static void WriteLine (string s)
{
  Contract.Requires (s != null);
  Console.WriteLine (s);
}
```

Because we're calling a method that requires the parameter to be non-null, we must prove that the argument is non-null. To do this, we can add a precondition to the first method as follows:

```
static void WriteLine (string s, bool b)
{
  Contract.Requires (s != null);
  if (b)
    WriteLine (s);     // OK
}
```

The ContractVerification Attribute

Static checking is easiest if instigated from the beginning of a project's lifecycle—otherwise you're likely to get overwhelmed with warnings.

If you do want to apply static contract checking to an existing codebase, it can help by initially applying it just to selective parts of a program—via the `ContractVerification` attribute (in `System.Diagnostics.Contracts`). This attribute can be applied at by initially applying it just to selective parts of a program—via the ContractVerification attribute (in System.Diagnostics.Contracts). This attribute can be applied at the assembly, type, and member level. If you apply it at multiple levels, the more granular wins. Therefore, to enable static contract verification just for a particular class, start by disabling verification at the assembly-level as follows:

```
[assembly: ContractVerification (false)]
```

and then enable it just for the desired class:

```
[ContractVerification (true)]
class Foo { ... }
```

Baselines

Another tactic in applying static contract verification to an existing codebase is to run the static checker with the *Baseline* option checked in Visual Studio. All the warnings that are produced are then written to a specified XML file. Next time you run static verification, all the warnings in that that file are ignored—so you see only messages generated as a result of *new* code that you've written.

messages generated as a result of *new* code that you've written.

The SuppressMessage Attribute

You can also tell the static checker to ignore certain types of warnings via the `SuppressMessage` attribute (in `System.Diagnostics.CodeAnalysis`):

```
    [SuppressMessage ("Microsoft.Contracts", warningFamily)]
```

where *`warningFamily`* is one of the following values:

```
Requires Ensures Invariant NonNull DivByZero MinValueNegation
ArrayCreation ArrayLowerBound ArrayUpperBound
```

You can apply this attribute at an assembly or type level.

Debugger Integration

Sometimes it's useful for an application to interact with a debugger if one is available. During development, the debugger is usually your IDE (e.g., Visual Studio); in deployment, the debugger is more likely to be:

-
-

DbgCLR

One of the lower-level debugging tools, such as WinDbg, Cordbg, or Mdbg

DbgCLR is Visual Studio stripped of everything but the debugger, and it is a free download with the .NET Framework SDK. It's the easiest debugging option when an IDE is not available, although it requires that you download the whole SDK.

Attaching and Breaking

The static `Debugger` class in `System.Diagnostics` provides basic functions for inter

The static `Debugger` class in `System.Diagnostics` provides basic functions for interacting with a debugger—namely `Break`, `Launch`, `Log`, and `IsAttached`.

A debugger must first attach to an application in order to debug it. If you start an application from within an IDE, this happens automatically, unless you request otherwise (by choosing “Start without debugging”). Sometimes, though, it’s inconvenient or impossible to start an application in debug mode within the IDE. An example is a Windows Service application or (ironically) a Visual Studio designer. One solution is to start the application normally, and then choose Debug Process in your IDE. This doesn’t allow you to set breakpoints early in the program’s execution, however.

The workaround is to call `Debugger.Break` from within your application. This method launches a debugger, attaches to it, and suspends execution at that point. (`Launch` does the same, but without suspending execution.) Once attached, you can log messages directly to the debugger’s output window with the `Log` method. You can tell whether you’re attached to a debugger with the `IsAttached` property.

Debugger Attributes

The DebuggerStepThrough and DebuggerHidden attributes provide suggestions to the debugger on how to handle single-stepping for a particular method, constructor, or class.

DebuggerStepThrough requests that the debugger step through a function without any user interaction. This attribute is useful in automatically generated methods and in proxy methods that forward the real work to a method somewhere else. In the latter proxy methods that forward the real work to a method somewhere else. In the latter case, the debugger will still show the proxy method in the call stack if a breakpoint is set within the "real" method—unless you also add the DebuggerHidden attribute. These two attributes can be combined on proxies to help the user focus on debugging the application logic rather than the plumbing:

```
[DebuggerStepThrough, DebuggerHidden]
void DoWorkProxy()
{
  // setup...
  DoWork();
  // teardown...
}

void DoWork() {...}   // Real method...
```

Processes and Process Threads

In the last section of Chapter 6, we described how to launch a new process with Process.Start. The Process class also allows you to query and interact with other processes running on the same, or another, computer.

Examining Running Processes

The `Process.GetProcess`*`XXX`* methods retrieve a specific process by name or process ID, or all processes running on the current or nominated computer. This includes both managed and unmanaged processes. Each `Process` instance has a wealth of properties mapping statistics such as name, ID, priority, memory and processor utilization, window handles, and so on. The following sample enumerates all the running processes on the current computer:

```
foreach (Process p in Process.GetProcesses())
using (p)
{
  Console.WriteLine (p.ProcessName);
  Console.WriteLine ("   PID:      " + p.Id);
  Console.WriteLine ("   Memory:   " + p.WorkingSet64);
  Console.WriteLine ("   Threads:  " + p.Threads.Count);
}
```

`Process.GetCurrentProcess` returns the current process. If you've created additional application domains, all will share the same process

`Process.GetCurrentProcess` returns the current process. If you've created additional application domains, all will share the same process.

You can terminate a process by calling its `Kill` method.

Examining Threads in a Process

You can also enumerate over the threads of other processes, with the `Process.Threads` property. The objects that you get, however, are not `System.Threading.Thread` objects, but rather `ProcessThread` objects, and are intended for administrative rather than synchronization tasks. A `ProcessThread` object provides diagnostic information about the underlying thread and allows you to control some aspects of it such as its priority and processor affinity:

```
public void EnumerateThreads (Process p)
{
  foreach (ProcessThread pt in p.Threads)
  {
    Console.WriteLine (pt.Id);
    Console.WriteLine ("   State:    " + pt.ThreadState);
    Console.WriteLine ("   Priority: " + pt.PriorityLevel);
    Console.WriteLine ("   Started:  " + pt.StartTime);
```

```
        Console.WriteLine ("   Priority: " + pt.PriorityLevel);
        Console.WriteLine ("   Started:  " + pt.StartTime);
        Console.WriteLine ("   CPU time: " + pt.TotalProcessorTime);
      }
    }
```

StackTrace and StackFrame

The StackTrace and StackFrame classes provide a read-only view of an execution call stack. You can obtain stack traces for the current thread, another thread in the same process, or an Exception object. Such information is useful mostly for diagnostic purposes, though it can also be used in programming (hacks). StackTrace represents a complete call stack; StackFrame represents a single method call within that stack.

If you instantiate a StackTrace object with no arguments—or with a bool argument—you get a snapshot of the current thread's call stack. The bool argument, if true, instructs StackTrace to read the assembly *.pdb* (project debug) files if they are present, giving you access to filename, line number, and column offset data.

Project debug files are generated when you compile with the /debug switch. Visual Studio compiles with this switch unless you request otherwise via *Advanced Build Settings*.

Once you've obtained a StackTrace, you can examine a particular frame by calling GetFrame—or obtain the whole lot with GetFrames:

```
static void Main() { A (); }
static void A()    { B (); }
static void B()    { C (); }
static void C()
{
  StackTrace s = new StackTrace (true);

 Console.WriteLine ("Total frames:   " + s.FrameCount);
 Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
```

```
  Console.WriteLine ("Total frames:   " + s.FrameCount);
  Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
  Console.WriteLine ("Calling method: " + s.GetFrame(1).GetMethod().Name);
  Console.WriteLine ("Entry method:   " + s.GetFrame
                                       (s.FrameCount-1).GetMethod().Name);
  Console.WriteLine ("Call Stack:");
  foreach (StackFrame f in s.GetFrames())
    Console.WriteLine (
      "  File: "   + f.GetFileName() +
      "  Line: "   + f.GetFileLineNumber() +
      "  Col: "    + f.GetFileColumnNumber() +
      "  Offset: " + f.GetILOffset() +
      "  Method: " + f.GetMethod().Name);
}
```

Diagnostics
Code Contr

Here's the output:

```
Total frames:   4
Current method: C
Calling method: B
Entry method: Main
Call stack:
  File: C:\Test\Program.cs
  File: C:\Test\Program.cs
  File: C:\Test\Program.cs
  File: C:\Test\Program.cs
  File: C:\Test\Program.cs

Line: 15
Line: 12
Line: 11
Line: 10

Col: 4  Offset: 7  Method: C
Col: 22  Offset: 6  Method: B
Col: 22  Offset: 6  Method: A
Col: 25  Offset: 6  Method: Main
```

```
Col: 25  Offset: 6  Method: Main
```

A shortcut to obtaining the essential information for an entire StackTrace is to call ToString on it. Here's what the result looks like:

```
   at DebugTest.Program.C() in C:\Test\Program.cs:line 16
   at DebugTest.Program.B() in C:\Test\Program.cs:line 12
   at DebugTest.Program.A() in C:\Test\Program.cs:line 11
   at DebugTest.Program.Main() in C:\Test\Program.cs:line 10
```

To obtain the stack trace for another thread, pass the other Thread into StackTrace's constructor. This can be a useful strategy for profiling a program. The one proviso is that you suspend the thread first, by calling its Suspend method (and Resume when you're done). This is the one valid use for Thread's deprecated Suspend and Resume methods!

You can also obtain the stack trace for an Exception object (showing what led up to the exception being thrown) by passing the Exception into StackTrace's constructor.

Exception already has a StackTrace property; however, this property returns a simple string—not a StackTrace object. A StackTrace object is far more useful in logging exceptions that occur after deployment—where no *.pdb* files are available—because you can log the *IL offset* in lieu of line and column numbers. With an IL offset and *ildasm*, you can pinpoint where within a method an error occurred.

Windows Event Logs

The Win32 platform provides a centralized logging mechanism, in the form of the Windows event logs.

The Debug and Trace classes we used earlier write to a Windows event log if you register an EventLogTraceListener. With the EventLog class, however, you can write directly to a Windows event log without using Trace or Debug. You can also use this class to read and monitor event data.

Writing to the Windows event log makes sense in a Windows Service application, because if something goes wrong, you can't pop up a user interface directing the user to some special file where diagnostic information has been written. Also, because it's common practice for services to write to the Windows event log, this is the first place an administrator is likely to look if your service falls over.

There are three standard Windows event logs, identified by these names:

- Application
- System
-

Security

The *Application* log is where most applications normally write.

Writing to the Event Log

To write to a Windows event log:

1. Choose one of the three event logs (usually *Application*).

2. Decide on a *source name* and create it if necessary.
3. Call `EventLog.WriteEntry` with the log name, source name, and message data.

The *source name* is an easily identifiable name for your application. You must register

The *source name* is an easily identifiable name for your application. You must register a source name before you use it—the CreateEventSource method performs this function. You can then call WriteEntry:

```
const string SourceName = "MyCompany.WidgetServer";
```

```
// CreateEventSource requires administrative permissions, so this would
// typically be done in application setup.
if (!EventLog.SourceExists (SourceName))
  EventLog.CreateEventSource (SourceName, "Application");
```

```
EventLog.WriteEntry (SourceName,
  "Service started; using configuration file=...",
  EventLogEntryType.Information);
```

EventLogEntryType can be Information, Warning, Error, SuccessAudit, or FailureAudit. Each displays with a different icon in the Windows event viewer. You can also optionally specify a category and event ID (each is a number of your own choosing) and provide optional binary data.

CreateEventSource also allows you to specify a machine name: this is to write to

CreateEventSource also allows you to specify a machine name: this is to write to another computer's event log, if you have sufficient permissions.

Reading the Event Log

To read an event log, instantiate the EventLog class with the name of the log you

To read an event log, instantiate the EventLog class with the name of the log you wish to access and optionally the name of another computer on which the log resides. Each log entry can then be read via the Entries collection property:

```
    EventLog log = new EventLog ("Application");

Console.WriteLine ("Total entries: " + log.Entries.Count);

EventLogEntry last = log.Entries [log.Entries.Count - 1];
Console.WriteLine ("Index:   " + last.Index);
Console.WriteLine ("Source:  " + last.Source);
Console.WriteLine ("Type:    " + last.EntryType);
Console.WriteLine ("Time:    " + last.TimeWritten);
Console.WriteLine ("Message: " + last.Message);
```

You can enumerate over all logs for the current (or another) computer with the static method EventLog.GetEventLogs (this requires administrative privileges):

```
    foreach (EventLog log in EventLog.GetEventLogs())
      Console.WriteLine (log.LogDisplayName);
```

This normally prints, at a minimum, *Application*, *Security*, and *System*.

This normally prints, at a minimum, *Application*, *Security*, and *System*.

Monitoring the Event Log

You can be alerted whenever an entry is written to a Windows event log, via the `EntryWritten` event. This works for event logs on the local computer, and it fires regardless of what application logged the event.

To enable log monitoring:

1. Instantiate an `EventLog` and set its `EnableRaisingEvents` property to `true`.
2. Handle the `EntryWritten` event.

For example:

```
static void Main()
static void Main()
{
  using (var log = new EventLog ("Application"))
  {
    log.EnableRaisingEvents = true;
    log.EntryWritten += DisplayEntry;
    Console.ReadLine();
  }
}
```

```
static void DisplayEntry (object sender, EntryWrittenEventArgs e)
{
  EventLogEntry entry = e.Entry;
  Console.WriteLine (entry.Message);
}
```

Performance Counters

Performance Counters

The logging mechanisms we've discussed to date are useful for capturing information for future analysis. However, to gain insight into the current state of an application (or the system as a whole), a more real-time approach is needed. The Win32 solution to this need is the performance-monitoring infrastructure, which consists of a set of performance counters that the system and applications expose, and the Microsoft Management Console (MMC) snap-ins used to monitor these counters in real time.

Performance counters are grouped into categories such as "System," "Processor," ".NET CLR Memory," and so on. These categories are sometimes also referred to as "performance objects" by the GUI tools. Each category groups a related set of performance counters that monitor one aspect of the system or application. Examples of performance counters in the ".NET CLR Memory" category include "% Time in GC," "# Bytes in All Heaps," and "Allocated bytes/sec."

Each category may optionally have one or more instances that can be monitored independently. For example, this is useful in the "% Processor Time" performance counter in the "Processor" category, which allows one to monitor CPU utilization. On a multiprocessor machine, this counter supports an instance for each CPU, allowing one to monitor the utilization of each CPU independently.

lowing one to monitor the utilization of each CPU independently.

The following sections illustrate how to perform commonly needed tasks, such as determining which counters are exposed, monitoring a counter, and creating your own counters to expose application status information.

Reading performance counters or categories may require administrator privileges on the local or target computer, depending on what is accessed.

Enumerating the Available Counters

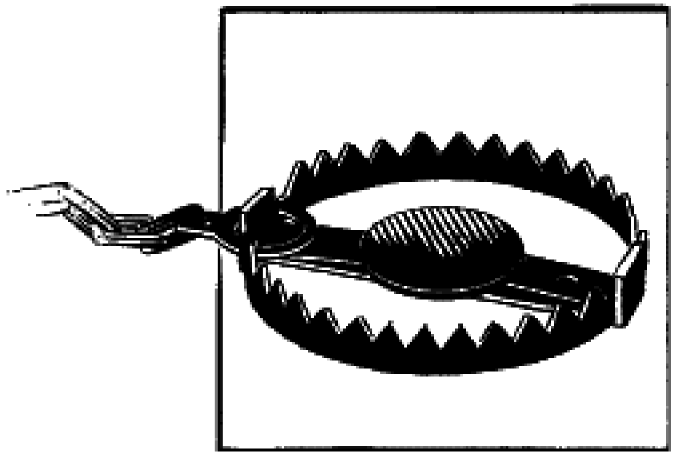
Enumerating the Available Counters

The following example enumerates over all of the available performance counters on the computer. For those that have instances, it enumerates the counters for each instance:

```
PerformanceCounterCategory[] cats =
  PerformanceCounterCategory.GetCategories();

foreach (PerformanceCounterCategory cat in cats)
{
  Console.WriteLine ("Category: " + cat.CategoryName);
```

```
  string[] instances = cat.GetInstanceNames();
  if (instances.Length == 0)
  {
    foreach (PerformanceCounter ctr in cat.GetCounters())
      Console.WriteLine ("  Counter: " + ctr.CounterName);
  }
  else   // Dump counters with instances
  {
```

```
  else   // Dump counters with instances
  {
    foreach (string instance in instances)
    {
      Console.WriteLine ("  Instance: " + instance);
      if (cat.InstanceExists (instance))
        foreach (PerformanceCounter ctr in cat.GetCounters (instance))
          Console.WriteLine ("    Counter: " + ctr.CounterName);
    }
  }
}
```



The result is more than 10,000 lines long! It also takes a while to execute because `PerformanceCounterCategory.InstanceExists` has an inefficient implementation. In a real system, you'd want to retrieve the more detailed information only on demand.

The next example uses a LINQ query to retrieve just .NET performance counters, writing the result to an XML file:

```
var x =
var x =
  new XElement ("counters",
    from PerformanceCounterCategory cat in
         PerformanceCounterCategory.GetCategories()
    where cat.CategoryName.StartsWith (".NET")
```

```
    let instances = cat.GetInstanceNames()
    select new XElement ("category",
      new XAttribute ("name", cat.CategoryName),
      instances.Length == 0
      ?
        from c in cat.GetCounters()
        select new XElement ("counter",
          new XAttribute ("name", c.CounterName))
      :
        from i in instances
        select new XElement ("instance", new XAttribute ("name", i),
          !cat.InstanceExists (i)
        select new XElement ("instance", new XAttribute ("name", i),
          !cat.InstanceExists (i)
          ?
            null
          :
            from c in cat.GetCounters (i)
            select new XElement ("counter",
              new XAttribute ("name", c.CounterName))
        )
      )
    );
x.Save ("counters.xml");
```

Reading Performance Counter Data

To retrieve the value of a performance counter, instantiate a `PerformanceCounter` object and then call the `NextValue` or `NextSample` method. `NextValue` returns a simple `float` value; `NextSample` returns a `CounterSample` object that exposes a more advanced set of properties, such as `CounterFrequency`, `TimeStamp`, `BaseValue`, and `RawValue`.

`PerformanceCounter`'s constructor takes a category name, counter name, and op-

PerformanceCounter's constructor takes a category name, counter name, and optional instance. So, to display the current processor utilization for all CPUs, you would do the following:

```
using (PerformanceCounter pc = new PerformanceCounter ("Processor",
                                                       "% Processor Time",
                                                       "_Total"))
  Console.WriteLine (pc.NextValue());
```

Or to display the "real" (i.e., private) memory consumption of the current process:

```
string procName = Process.GetCurrentProcess().ProcessName;
using (PerformanceCounter pc = new PerformanceCounter ("Process",
                                                       "Private Bytes",
                                                       procName))
  Console.WriteLine (pc.NextValue());
```

PerformanceCounter doesn't expose a ValueChanged event, so if you want to monitor for changes, you must poll. In the next example, we poll every 200 ms—until signaled to quit by an EventWaitHandle:

```
// need to import System.Threading as well as System.Diagnostics
```

```
static void Monitor (string category, string counter, string instance,
                     EventWaitHandle stopper)
```

```
                        EventWaitHandle stopper)
```

```
{
  if (!PerformanceCounterCategory.Exists (category))
    throw new InvalidOperationException ("Category does not exist");

  if (!PerformanceCounterCategory.CounterExists (counter, category))
    throw new InvalidOperationException ("Counter does not exist");

  if (instance == null) instance = "";   // "" == no instance (not null!)
  if (instance != "" &&
      !PerformanceCounterCategory.InstanceExists (instance, category))
    throw new InvalidOperationException ("Instance does not exist");

  float lastValue = 0f;
  using (PerformanceCounter pc = new PerformanceCounter (category,
                                                      counter, instance))
    while (!stopper.WaitOne (200, false))
    {
      float value = pc.NextValue();
      {
        float value = pc.NextValue();
        if (value != lastValue)          // Only write out the value
        {                                // if it has changed.
          Console.WriteLine (value);
          lastValue = value;
        }
      }
}
```

Here's how we can use this method to simultaneously monitor processor and hard-disk activity:

```
    static void Main()
    {
      EventWaitHandle stopper = new ManualResetEvent (false);
```

```
new Thread (() =>
  Monitor ("Processor", "% Processor Time", "_Total", stopper)
).Start();

new Thread (() =>
  Monitor ("LogicalDisk", "% Idle Time", "C:", stopper)
).Start();

  Console.WriteLine ("Monitoring - press any key to quit");
  Console.ReadKey();
  stopper.Set();
}
```

```
}
```

Creating Counters and Writing Performance Data

Before writing performance counter data, you need to create a performance category and counter. You must create the performance category along with all the counters that belong to it in one step, as follows:

```
string category = "Nutshell Monitoring";
```

```
// We'll create two counters in this category:
```

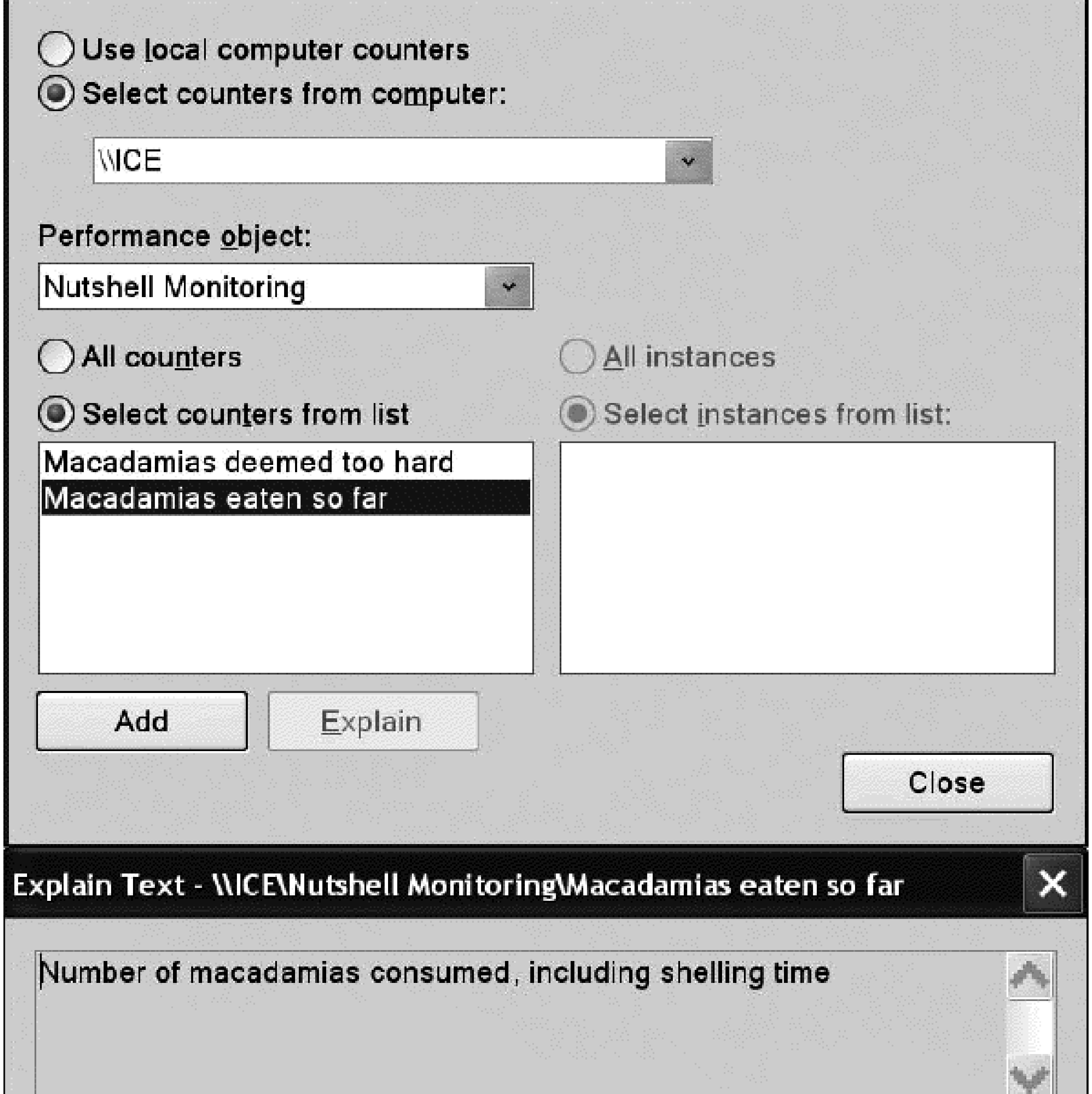
```
string eatenPerMin = "Macadamias eaten so far";
string tooHard = "Macadamias deemed too hard";
```

```
if (!PerformanceCounterCategory.Exists (category))
{
  CounterCreationDataCollection cd = new CounterCreationDataCollection();
```

```
{
  CounterCreationDataCollection cd = new CounterCreationDataCollection();

cd.Add (new CounterCreationData (eatenPerMin,
        "Number of macadamias consumed, including shelling time",
        PerformanceCounterType.NumberOfItems32));

cd.Add (new CounterCreationData (tooHard,
        "Number of macadamias that will not crack, despite much effort",
        PerformanceCounterType.NumberOfItems32));

  PerformanceCounterCategory.Create (category, "Test Category",
    PerformanceCounterCategoryType.SingleInstance, cd);
}
```

The new counters then show up in the Windows performance-monitoring tool when you choose Add Counters, as shown in Figure 13-1.

Use local computer counters


( ) Use local computer counters
[● Select counters from computer:]

\\ICE

Performance object:

Nutshell Monitoring

( ) All counters
[● Select counters from list]

( ) All instances
[● Select instances from list:]

Macadamias deemed too hard
Macadamias eaten so far

Explain Text - \\ICE\Nutshell Monitoring\Macadamias eaten so far

Number of macadamias consumed, including shelling time

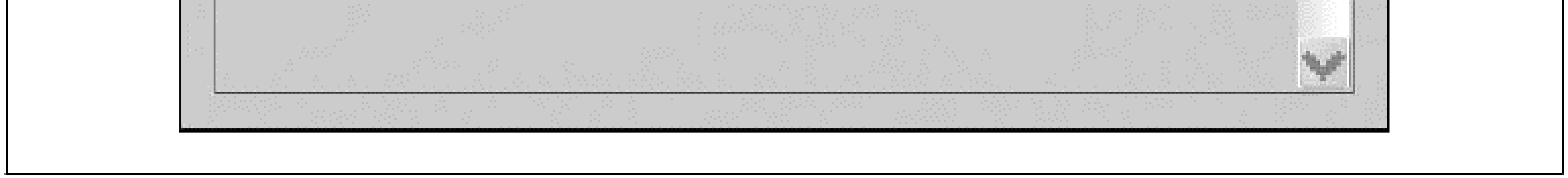


Figure 13-1. Custom performance counter

If you later want to define more counters in the same category, you must first delete the old category by calling `PerformanceCounterCategory.Delete`.

Creating and deleting performance counters requires administrative privileges. For this reason, it's usually done as part of the application setup.

Once a counter is created, you can update its value by instantiating a `PerformanceCounter`, setting `ReadOnly` to `false`, and setting `RawValue`. You can also use the `Increment` and `IncrementBy` methods to update the existing value:

```
string category = "Nutshell Monitoring";
string category = "Nutshell Monitoring";
string eatenPerMin = "Macadamias eaten so far";
```

```
using (PerformanceCounter pc = new PerformanceCounter (category,
                                                      eatenPerMin, ""))
{
  pc.ReadOnly = false;
  pc.RawValue = 1000;
  pc.Increment();
  pc.IncrementBy (10);
  Console.WriteLine (pc.NextValue());    // 1011
}
```

The Stopwatch Class

The Stopwatch class provides a convenient mechanism for measuring execution times. Stopwatch uses the highest-resolution mechanism that the operating system and hardware provide, which is typically less than a microsecond. (In contrast, DateTime.Now and Environment.TickCount have a resolution of about 15 ms.)

To use Stopwatch, call StartNew—this instantiates a Stopwatch and starts it ticking. (Alternatively, you can instantiate it manually and then call Start.) The Elapsed property returns the elapsed interval as a TimeSpan:

```
Stopwatch s = Stopwatch.StartNew();
System.IO.File.WriteAllText ("test.txt", new string ('*', 30000000));
Console.WriteLine (s.Elapsed);       // 00:00:01.4322661
```

```
    Console.WriteLine (s.Elapsed);        // 00:00:01.4322661
```

`Stopwatch` also exposes an `ElapsedTicks` property, which returns the number of elapsed “ticks” as a `long`. To convert from ticks to seconds, divide by `StopWatch.Frequency`. There’s also an `ElapsedMilliseconds` property, which is often the most convenient.

Calling `Stop` freezes `Elapsed` and `ElapsedTicks`. There’s no background activity incurred by a “running” `Stopwatch`, so calling `Stop` is optional.

1

4 Streams and I/O

This chapter describes the fundamental types for input and output in .NET, with emphasis on the following topics:

-
-
-

•

•

The .NET stream architecture and how it provides a consistent programming interface for reading and writing across a variety of I/O types

Manipulating files and directories on disk

Isolated storage and its role in segregating data by program and user

This chapter concentrates on the types in the `System.IO` namespace, the home of lower-level I/O functionality. The .NET Framework also provides higher-level I/O functionality in the form of SQL connections and commands, LINQ to SQL and LINQ to XML, Windows Communication Foundation, Web Services, and Remoting.

Stream Architecture

The .NET stream architecture centers on three concepts: backing stores, decorators, and adapters, as shown in Figure 14-1.

A *backing store* is the endpoint that makes input and output useful, such as a file or

A *backing store* is the endpoint that makes input and output useful, such as a file or network connection. Precisely, it is either or both of the following:

-
-

A source from which bytes can be sequentially read

A destination to which bytes can be sequentially written

A backing store is of no use, though, unless exposed to the programmer. A `Stream` is the standard .NET class for this purpose; it exposes a standard set of methods for reading, writing, and positioning. Unlike an array, where all the backing data exists in memory at once, a stream deals with data serially—either one byte at a time or in blocks of a manageable size. Hence, a stream can use little memory regardless of the size of its backing store.

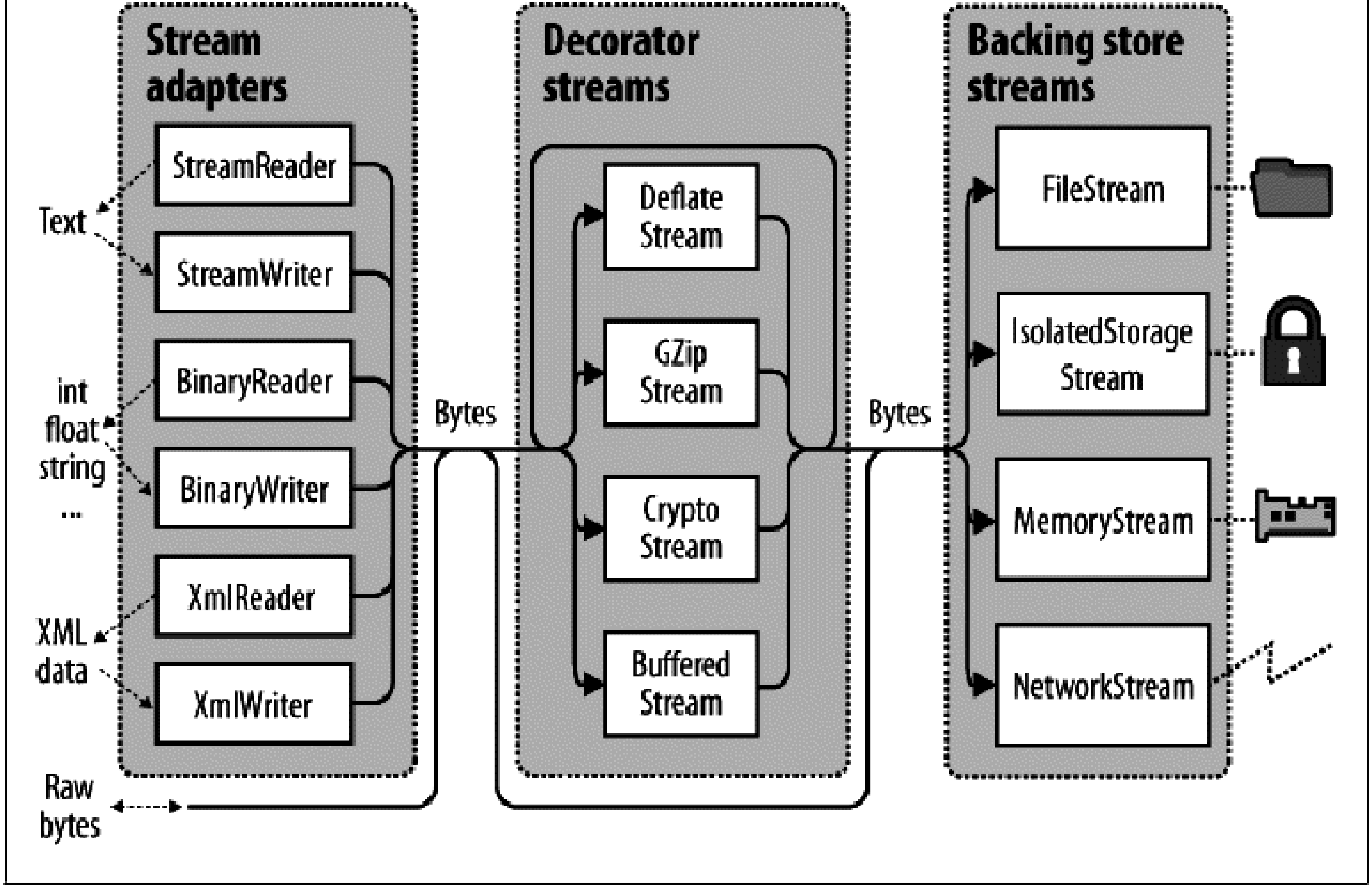


Figure 14-1. Stream architecture

Streams fall into two categories:

Streams fall into two categories:

Backing store streams
: These are hard-wired to a particular type of backing store, such as `FileStream` or `NetworkStream`

Decorator streams
: These feed off another stream, transforming the data in some way, such as `DeflateStream` or `CryptoStream`

Decorator streams have the following architectural benefits:

-
-
-
-

They liberate backing store streams from needing to implement such features as compression and encryption themselves.

Streams don’t suffer a change of interface when decorated.

Streams don't suffer a change of interface when decorated.

You connect decorators at runtime.

You can chain decorators together (e.g., a compressor followed by an encryptor).

Both backing store and decorator streams deal exclusively in bytes. Although this is flexible and efficient, applications often work at higher levels such as text or XML. *Adapters* bridge this gap by wrapping a stream in a class with specialized methods typed to a particular format. For example, a text reader exposes a `ReadLine` method; an XML writer exposes a `WriteAttributes` method.

An adapter wraps a stream, just like a decorator. Unlike a decorator, however, an adapter is not *itself* a stream; it typically hides the byte-oriented methods completely.

To summarize, backing store streams provide the raw data; decorator streams provide transparent binary transformations such as encryption; adapters offer typed

To summarize, backing store streams provide the raw data; decorator streams provide transparent binary transformations such as encryption; adapters offer typed methods for dealing in higher-level types such as strings and XML. Figure 14-1 illustrates their associations. To compose a chain, you simply pass one object into another's constructor.

Using Streams

The abstract `Stream` class is the base for all streams. It defines methods and properties for three fundamental operations, *reading*, *writing*, and *seeking*, as well as for administrative tasks such as closing, flushing, and configuring timeouts (see Table 14-1).

Table 14-1. Stream class members

Category	Members

Reading	`public abstract bool CanRead { get; }`
	`public abstract int Read (byte[] buffer, int offset, int count)`
	`public virtual int ReadByte();`

Writing	`public abstract bool CanWrite { get; }`

	`public virtual int ReadByte();`

Writing	`public abstract bool CanWrite { get; }`
	`public abstract void Write (byte[] buffer, int offset, int count);`
	`public virtual void WriteByte (byte value);`

Seeking	`public abstract bool CanSeek { get; }`
	`public abstract long Position { get; set; }`
	`public abstract void SetLength (long value);`
	`public abstract long Length { get; }`
	`public abstract long Seek (long offset, SeekOrigin origin);`

Closing/flushing	`public virtual void Close();`
	`public void Dispose();`
	`public abstract void Flush();`

Timeouts	`public virtual bool CanTimeout { get; }`
	`public virtual int ReadTimeout { get; set; }`
	`public virtual int WriteTimeout { get; set; }`

Other	`public static readonly Stream Null; // "Null" stream`
	`public static Stream Synchronized (Stream stream);`

In the following example, we use a file stream to read, write, and seek:

```
using System;
using System.IO;
using System.IO;
```

```
class Program
{
  static void Main()
  {
    // Create a file called test.txt in the current directory:
    using (Stream s = new FileStream ("test.txt", FileMode.Create))
    {
      Console.WriteLine (s.CanRead);       // True
      Console.WriteLine (s.CanWrite);      // True
      Console.WriteLine (s.CanSeek);       // True
```

```
s.WriteByte (101);
s.WriteByte (102);
byte[] block = { 1, 2, 3, 4, 5 };
s.Write (block, 0, block.Length);
```

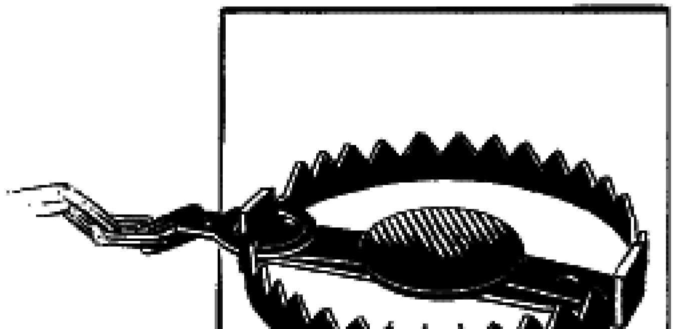
```
s.Write (block, 0, block.Length);
Console.WriteLine (s.Length);
Console.WriteLine (s.Position);
s.Position = 0;
Console.WriteLine (s.ReadByte());
Console.WriteLine (s.ReadByte());
```

```
// Write block of 5 bytes
// 7
// 7
// Move back to the start
// Move back to the start
// 101
// 102

// Read from the stream back into the block array:
Console.WriteLine (s.Read (block, 0, block.Length));

// 5
// 0

      // Assuming the last Read returned 5, we'll be at
      // the end of the file, so Read will now return 0:
      Console.WriteLine (s.Read (block, 0, block.Length));
    }
  }
}
```

Reading and Writing

A stream may support reading, writing, or both. If `CanWrite` returns `false`, the stream is read-only; if `CanRead` returns `false`, the stream is write-only.

`Read` receives a block of data from the stream into an array. It returns the number of bytes received, which is always either less than or equal to the `count` argument. If it's less than `count`, it means either that the end of the stream has been reached or the stream is giving you the data in smaller chunks (as is often the case with network streams). In either case, the balance of bytes in the array will remain unwritten, their previous values preserved.



With `Read`, you can be certain you've reached the end of the stream only when the method returns `0`. So, if you have a 1,000-byte stream, the following code may fail to read it all into memory:

```
// Assuming s is a stream:
byte[] data = new byte [1000];
s.Read (data, 0, data.Length);
```

The `Read` method could read anywhere from 1 to 1,000 bytes, leaving the balance of the stream unread.

Here's the correct way to read a 1,000-byte stream:

```
byte[] data = new byte [1000];
```

```
byte[] data = new byte [1000];

// bytesRead will always end up at 1000, unless the stream is
// itself smaller in length:

int bytesRead = 0;
int chunkSize = 1;
while (bytesRead < data.Length && chunkSize > 0)
  bytesRead +=
    chunkSize = s.Read (data, bytesRead, data.Length - bytesRead);
```

Fortunately, the `BinaryReader` type provides a simpler way to achieve the same result:

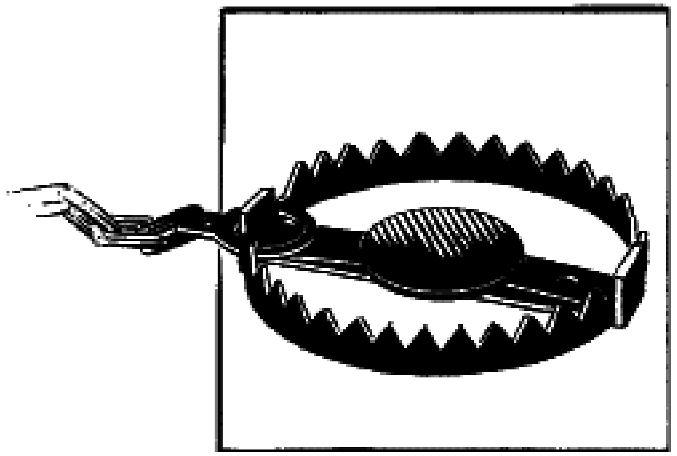
```
byte[] data = new BinaryReader (s).ReadBytes (1000);
```

If the stream is less than 1,000 bytes long, the byte array returned reflects the actual stream size. If the stream is seekable, you can read its entire contents by replacing `1000` with `(int)s.Length`.

We describe the `BinaryReader` type further in the section "Stream Adapters" on page 552, later in this chapter.

The `ReadByte` method is simpler: it reads just a single byte, returning –1 to indicate the end of the stream. `ReadByte` actually returns an `int` rather than a `byte`, as the latter cannot return –1.

The `Write` and `WriteByte` methods send data to the stream. If they are unable to send the specified bytes, an exception is thrown.



In the `Read` and `Write` methods, the `offset` argument refers to the index in the `buffer` array at which reading or writing begins, not the position within the stream.

Streams also support asynchronous reading and writing through the methods `BeginRead` and `BeginWrite`. Asynchronous methods are intended for high-throughput server applications, and we describe them in Chapter 23.

Seeking

A stream is seekable if `CanSeek` returns `true`. With a seekable stream (such as a file stream), you can query or modify its `Length` (by calling `SetLength`), and at any time change the `Position` at which you're reading or writing. The `Position` property is relative to the beginning of the stream; the `Seek` method, however, allows you to move relative to the current position or the end of the stream.

Changing the `Position` on a `FileStream` typically takes a few microseconds. If you're doing this millions of times in a loop, Framework 4.0's new `MemoryMappedFile` class may be a better choice than a `FileStream` (for more, see "Memory-Mapped Files" on page 569, later in this chapter).

With a nonseekable stream (such as an encryption stream), the only way to determine its length is to read it right through. Furthermore, if you need to reread a previous section, you must close the stream and start afresh with a new one.

mine its length is to read it right through. Furthermore, if you need to reread a previous section, you must close the stream and start afresh with a new one.

Closing and Flushing

Streams must be disposed after use to release underlying resources such as file and socket handles. A simple way to guarantee this is by instantiating streams within `using` blocks. In general, streams follow standard disposal semantics:

•

•

`Dispose` and `Close` are identical in function.

Disposing or closing a stream repeatedly causes no error.

Closing a decorator stream closes both the decorator and its backing store stream. With a chain of decorators, closing the outermost decorator (at the head of the chain) closes the whole lot.

Some streams internally buffer data to and from the backing store to lessen round-

Some streams internally buffer data to and from the backing store to lessen round-tripping and so improve performance (file streams are a good example of this). This means data you write to a stream may not hit the backing store immediately; it can be delayed as the buffer fills up. The `Flush` method forces any internally buffered data to be written immediately. `Flush` is called automatically when a stream is closed, so you never need to do the following:

```
    s.Flush(); s.Close();
```

Timeouts

A stream supports read and write timeouts if `CanTimeout` returns `true`. Network streams support timeouts; file and memory streams do not. For streams that support timeouts, the `ReadTimeout` and `WriteTimeout` properties determine the desired timeout in milliseconds, where 0 means no timeout. The `Read` and `Write` methods indicate that a timeout has occurred by throwing an exception.

Thread Safety

Thread Safety

As a rule, streams are not thread-safe, meaning that two threads cannot concurrently read or write to the same stream without possible error. The `Stream` class offers a simple workaround via the static `Synchronized` method. This method accepts a stream of any type and returns a thread-safe wrapper. The wrapper works by obtaining an exclusive lock around each read, write, or seek, ensuring that only one thread can perform such an operation at a time. In practice, this allows multiple threads to simultaneously append data to the same stream—other kinds of activities (such as concurrent reading) require additional locking to ensure that each thread accesses the desired portion of the stream. We discuss thread safety fully in Chapter 21.

Backing Store Streams

Figure 14-2 shows the key backing store streams provided by the .NET Framework. A "null stream" is also available, via the `Stream`'s static `Null` field.

In the following sections, we describe `FileStream` and `MemoryStream`; in the final sec-

In the following sections, we describe `FileStream` and `MemoryStream`; in the final section in this chapter, we describe `IsolatedStorageStream`. In Chapter 15, we cover `NetworkStream`.

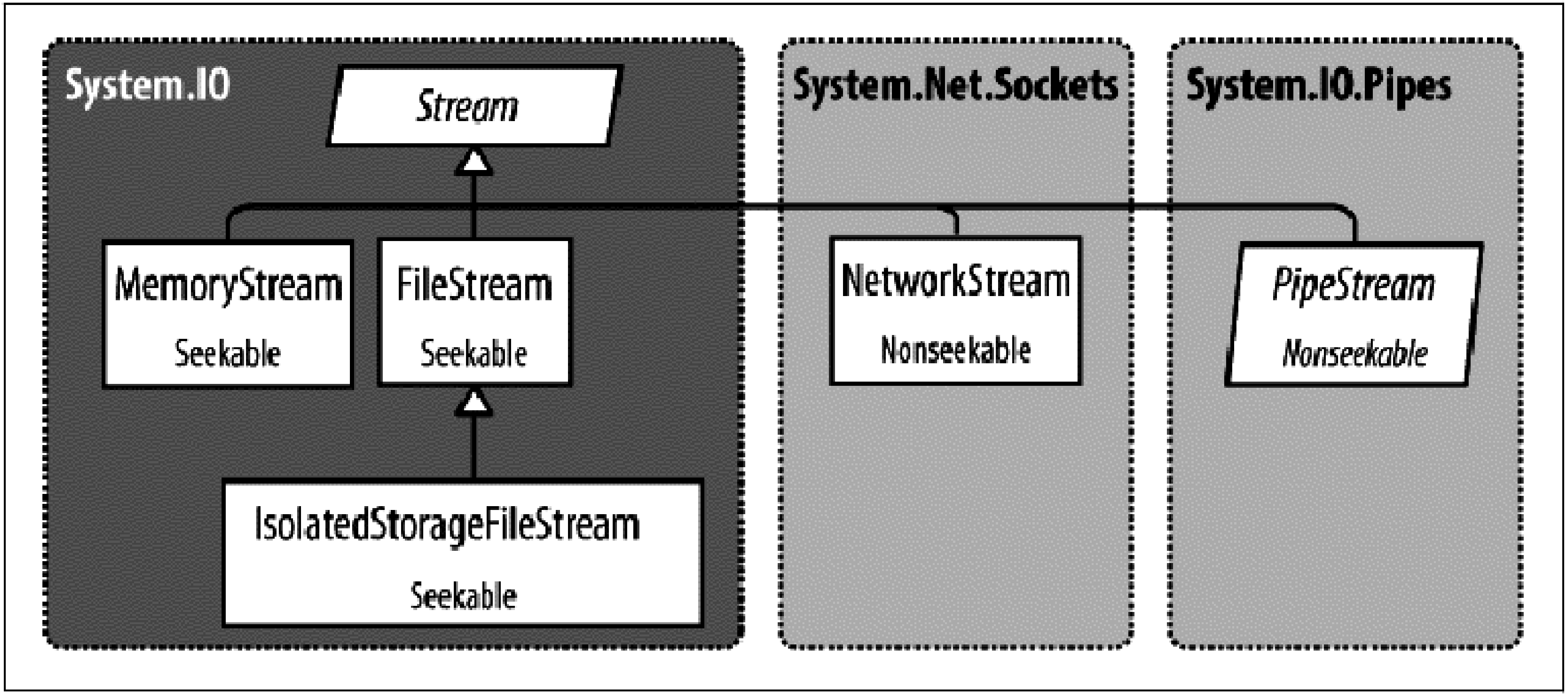


Figure 14-2. Backing store streams

FileStream

Earlier in this section, we demonstrated the basic use of a `FileStream` to read and write bytes of data. We'll now examine the special features of this class.

Constructing a FileStream

The simplest way to instantiate a `FileStream` is to use one of the following static

The simplest way to instantiate a FileStream is to use one of the following static façade methods on the File class:

```
FileStream fs1 = File.OpenRead  ("readme.bin");            // Read-only
FileStream fs2 = File.OpenWrite (@"c:\temp\writeme.tmp");  // Write-only
FileStream fs3 = File.Create    (@"c:\temp\writeme.tmp");  // Read/write
```

OpenWrite and Create differ in behavior if the file already exists. Create truncates any existing content; OpenWrite leaves existing content intact with the stream positioned at zero. If you write fewer bytes than were previously in the file, OpenWrite leaves you with a mixture of old and new content.

You can also instantiate a FileStream directly. Its constructors provide access to every feature, allowing you to specify a filename or low-level file handle, file creation and access modes, and options for sharing, buffering, and security. The following opens an existing file for read/write access without overwriting it:

```
var fs = new FileStream ("readwrite.tmp", FileMode.Open);  // Read/write
```

More on FileMode shortly.

More on `FileMode` shortly.

Shortcut Methods on the File Class

The following static methods read an entire file into memory in one step:

- `File.ReadAllText` (returns a string)
- `File.ReadAllLines` (returns an array of strings)
- `File.ReadAllBytes` (returns a byte array)

The following static methods write an entire file in one step:

- `File.WriteAllText`
- `File.WriteAllLines`
- `File.WriteAllBytes`
- `File.AppendAllText` (great for appending to a log file)

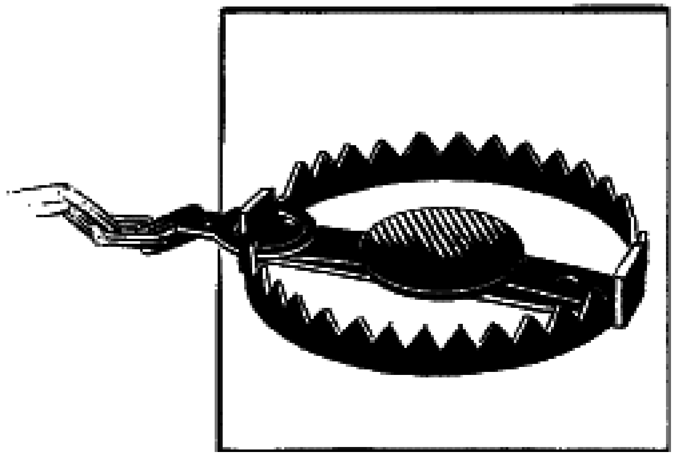
From Framework 4.0 there's also a static method called `File.ReadLines`: this is like `ReadAllLines` except that it returns a lazily-evaluated `IEnumerable<string>`. This is more efficient because it doesn't load the entire file into memory at once. LINQ is ideal for consuming the results: the following calculates the number of lines greater than 80 characters in length:

lines greater than 80 characters in length:

```
    int longLines = File.ReadLines ("filePath")
                        .Count (l => l.Length > 80);
```

Specifying a filename

A filename can be either absolute (e.g., *c:\temp\test.txt*) or relative to the current directory (e.g., *test.txt* or *temp\test.txt*). You can access or change the current directory via the static `Environment.CurrentDirectory` property.



When a program starts, the current directory may or may not coincide with that of the program's executable. For this reason, When a program starts, the current directory may or may not coincide with that of the program's executable. For this reason, you should never rely on the current directory for locating additional runtime files packaged along with your executable.

`AppDomain.CurrentDomain.BaseDirectory` returns the *application base directory*, which in normal cases is the folder containing the program's executable. To specify a filename relative to this directory, you can call `Path.Combine`:

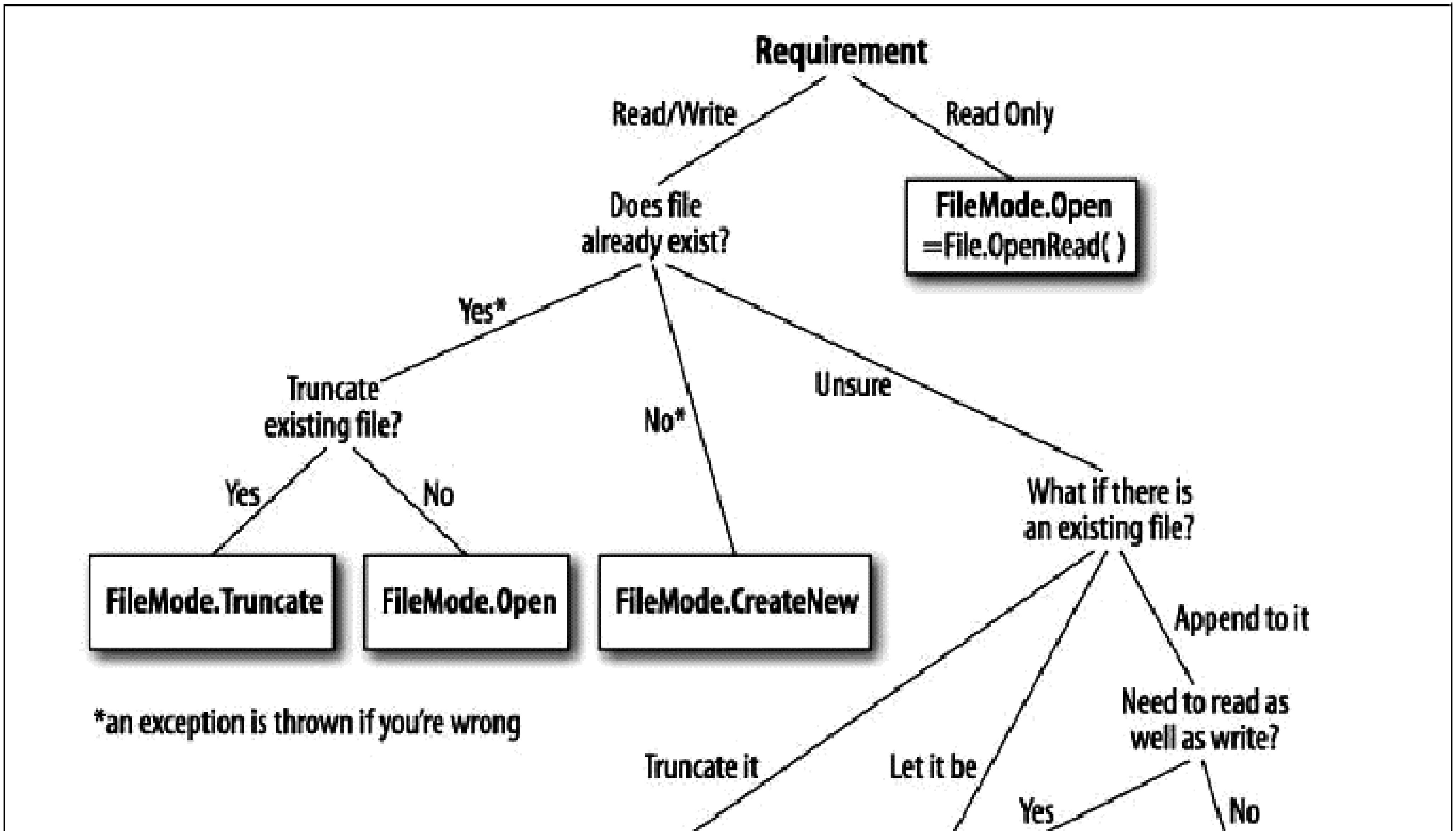
```
string baseFolder = AppDomain.CurrentDomain.BaseDirectory;
string logoPath = Path.Combine (baseFolder, "logo.jpg");
Console.WriteLine (File.Exists (logoPath));
```

You can read and write across a network via a UNC path, such as *\\JoesPC\PicShare\pic.jpg* or *\\10.1.1.2\PicShare\pic.jpg*.

Specifying a FileMode

Specifying a FileMode

All of `FileStream`'s constructors that accept a filename also require a `FileMode` enum argument. Figure 14-3 shows how to choose a `FileMode`, and the choices yield results akin to calling a static method on the `File` class.



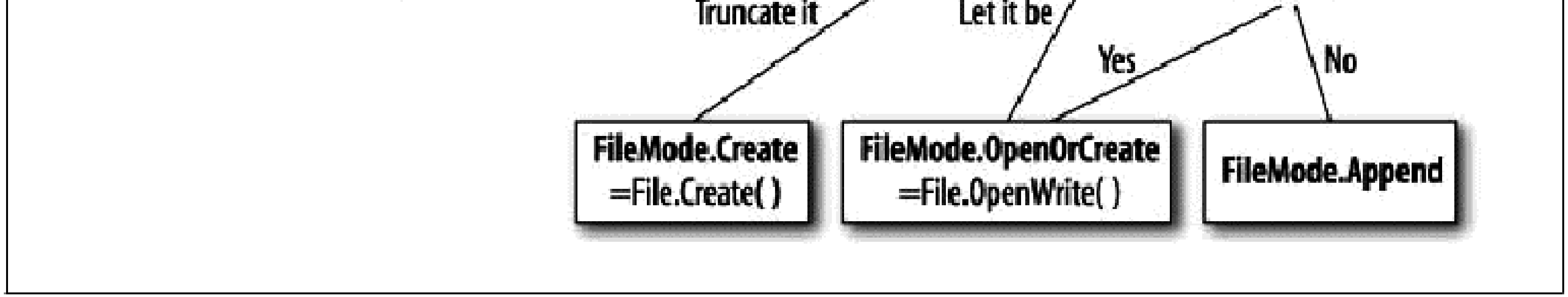
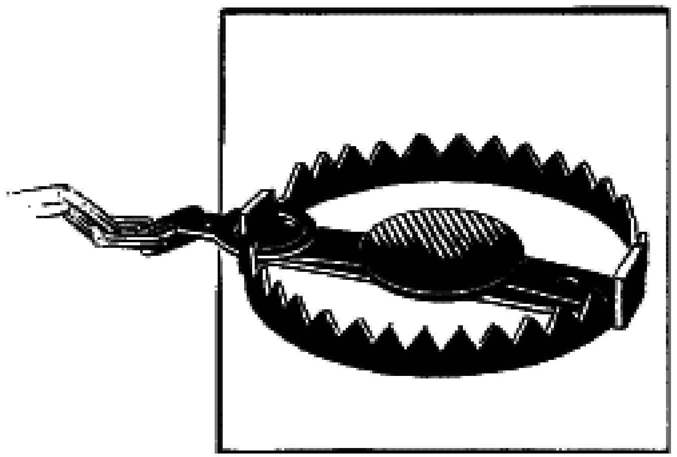

Truncate it
Let it be
Yes
No
FileMode.Create =File.Create( )
FileMode.OpenOrCreate =File.OpenWrite( )
FileMode.Append

Figure 14-3. Choosing a FileMode



`File.Create` and `FileMode.Create` will throw an exception if used on hidden files. To overwrite a hidden file, you must delete and re-create it:

```
if (File.Exists ("hidden.txt")) File.Delete ("hidden.txt");
```

Constructing a `FileStream` with just a filename and `FileMode` gives you (with just one exception) a readable writable stream. You can request a downgrade if you also supply a `FileAccess` argument:

[Flags]

```
[Flags]
public enum FileAccess { Read = 1, Write = 2, ReadWrite = 3 }
```

The following returns a read-only stream, equivalent to calling `File.OpenRead`:

```
    using (var fs = new FileStream ("x.bin", FileMode.Open, FileAccess.Read))
      ...
```

`FileMode.Append` is the odd one out: with this mode, you get a *write-only* stream. To append with read-write support, you must instead use `FileMode.Open` or `FileMode.OpenOrCreate`, and then seek the end of the stream:

```
    using (var fs = new FileStream ("myFile.bin", FileMode.Open))
    {
      fs.Seek (0, SeekOrigin.End);
      ...
```

Advanced FileStream features

Here are other optional arguments you can include when constructing a `FileStream`:

-
-

●

●

●

●

●

A FileShare enum describing how much access to grant other processes wanting to dip into the same file before you've finished (None, Read [default], ReadWrite, or Write).

The size, in bytes, of the internal buffer (default is currently 4 KB).

A flag indicating whether to defer to the operating system for asynchronous I/O.

A FileSecurity object describing what user and role permissions to assign a new file.

A FileOptions flags enum for requesting operating system encryption (Encrypted), automatic deletion upon closure for temporary files (DeleteOnClose), and optimization hints (RandomAccess and SequentialScan). There is also a WriteThrough flag that requests that the operating system disable write-behind caching; this is for transactional files or logs.

Opening a file with FileShare.ReadWrite allows other processes or users to simul-

Opening a file with `FileShare.ReadWrite` allows other processes or users to simultaneously read and write to the same file. To avoid chaos, you can all agree to lock specified portions of the file before reading or writing, using these methods:

```
// Defined on the FileStream class:
public virtual void Lock   (long position, long length);
public virtual void Unlock (long position, long length);
```

`Lock` throws an exception if part or all of the requested file section has already been locked. This is the system used in file-based databases such as Access and FoxPro.

MemoryStream

`MemoryStream` uses an array as a backing store. This partly defeats the purpose of having a stream, because the entire backing store must reside in memory at once. `MemoryStream` still has uses, however; an example is when you need random access to a nonseekable stream. If you know the source stream will be of a manageable size, you can copy it into a `MemoryStream` as follows:

```
static MemoryStream ToMemoryStream (this Stream input, bool closeInput)
{
  try
  {                                           // Read and write in
    byte[] block = new byte [0x1000];         // blocks of 4K.
    MemoryStream ms = new MemoryStream();
    while (true)
    {
      int bytesRead = input.Read (block, 0, block.Length);
      if (bytesRead == 0) return ms;
      ms.Write (block, 0, bytesRead);
    }
  }
  finally { if (closeInput) input.Close (); }
}
```

The reason for the closeInput argument is to avoid a situation where the method author and consumer each think the other will close the stream.

You can convert a MemoryStream to a byte array by calling ToArray. The GetBuffer method does the same job more efficiently by returning a direct reference to the method does the same job more efficiently by returning a direct reference to the underlying storage array; unfortunately, this array is usually longer than the stream's real length.

> Closing and flushing a `MemoryStream` is optional. If you close a `MemoryStream`, you can no longer read or write to it, but you are still permitted to call `ToArray` to obtain the underlying data. `Flush` does absolutely nothing on a memory stream.

You can find further `MemoryStream` examples in the section "Compression" on page 571 later in this chapter, and in the section "Cryptography Overview" on page 776 in Chapter 20.

PipeStream

`PipeStream` was introduced in Framework 3.5. It provides a simple means by which one process can communicate with another through the Windows *pipes* protocol. There are two kinds of pipe:

Anonymous pipe
: Allows one-way communication between a parent and child process on the same computer.id

Named pipe
: Allows two-way communication between arbitrary processes on the same computer—or different computers across a Windows network.