```
{
  static ConcurrentQueue<TempFileRef> _failedDeletions
    = new ConcurrentQueue<TempFileRef>();

public readonly string FilePath;
public Exception DeletionError { get; private set; }

public TempFileRef (string filePath) { FilePath = filePath; }

   ~TempFileRef()
   {
     try { File.Delete (FilePath); }
     catch (Exception ex)
     {
       DeletionError = ex;
        failedDeletions.Enqueue (this);
```

```
            _failedDeletions.Enqueue (this);
        }
    }
}
// Resurrection
```

Enqueuing the object to the static `_failedDeletions` collection gives the object another referee, ensuring that it remains alive until the object is eventually dequeued.

`ConcurrentQueue<T>` is a thread-safe version of `Queue<T>` and is defined in `System.Collections.Concurrent` (see Chapter 22). There are a couple of reasons for using a thread-safe collection. First, the CLR reserves the right to execute finalizers on more than one thread in parallel. This means that when accessing shared state such as a static collection, we must consider the possibility of two objects being finalized at once. Second, at shared state such as a static collection, we must consider the possibility of two objects being finalized at once. Second, at some point we're going to want to dequeue items from `_failedDeletions` so that we can do something about them. This also has to be done in a thread-safe fashion, because it could happen while the finalizer is concurrently enqueuing another object.

GC.ReRegisterForFinalize

A resurrected object's finalizer will not run a second time—unless you call `GC.ReRegisterForFinalize`.

In the following example, we try to delete a temporary file in a finalizer (as in the last example). But if the deletion fails, we reregister the object so as to try again in the next garbage collection:

```
public class TempFileRef
{
```

```
public class TempFileRef
{
  public readonly string FilePath;
  int _deleteAttempt;

public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef()
  {
    try { File.Delete (FilePath); }
    catch
    {
      if (_deleteAttempt++ < 3) GC.ReRegisterForFinalize (this);
    }
  }
}
```

After the third failed attempt, our finalizer will silently give up trying to delete the file. We could enhance this by combining it with the previous example—in other

After the third failed attempt, our finalizer will silently give up trying to delete the file. We could enhance this by combining it with the previous example—in other words, adding it to the `_failedDeletions` queue after the third failure.

Be careful to call `ReRegisterForFinalize` just once in the finalizer method. If you call it twice, the object will be reregistered twice and will have to undergo two more finalizations!

and will have to undergo two more finalizations!

How the Garbage Collector Works

The CLR uses a generational mark-and-compact GC that performs automatic memory management for objects stored on the managed heap. The GC is considered to be a *tracing* garbage collector in that it doesn't interfere with every access to an object, but rather wakes up intermittently and traces the graph of objects stored on the managed heap to determine which objects can be considered garbage and therefore collected.

The GC initiates a garbage collection upon performing a memory allocation (via the `new` keyword) either after a certain threshold of memory has been allocated, or at other times to reduce the application's memory footprint. This process can also be initiated manually by calling `System.GC.Collect`. During a garbage collection, all threads may by frozen (more on this in the next section).

The GC begins with its root object references, and walks the object graph, marking all the objects it touches as reachable. Once this process is complete, all objects that have not been marked are considered unused, and are subject to garbage collection.

have not been marked are considered unused, and are subject to garbage collection.

Unused objects without finalizers are immediately discarded; unused objects with finalizers are enqueued for processing on the finalizer thread after the GC is complete. These objects then become eligible for collection in the next GC for the object's generation (unless resurrected).

The remaining "live" objects are then shifted to the start of the heap (compacted), freeing space for more objects. This compaction serves two purposes: it avoids memory fragmentation, and it allows the GC to employ a very simple strategy when allocating new objects, which is to always allocate memory at the end of the heap. This avoids the potentially time-consuming task of maintaining a list of free memory segments.

If there is insufficient space to allocate memory for a new object after garbage collection, and the operating system is unable to grant further memory, an `OutOfMemoryException` is thrown.

Optimization Techniques

Optimization Techniques

The GC incorporates various optimization techniques to reduce the garbage collection time.

Generational collection

The most important optimization is that the GC is generational. This takes advantage of the fact that although many objects are allocated and discarded rapidly, certain objects are long-lived and thus don’t need to be traced during every collection.

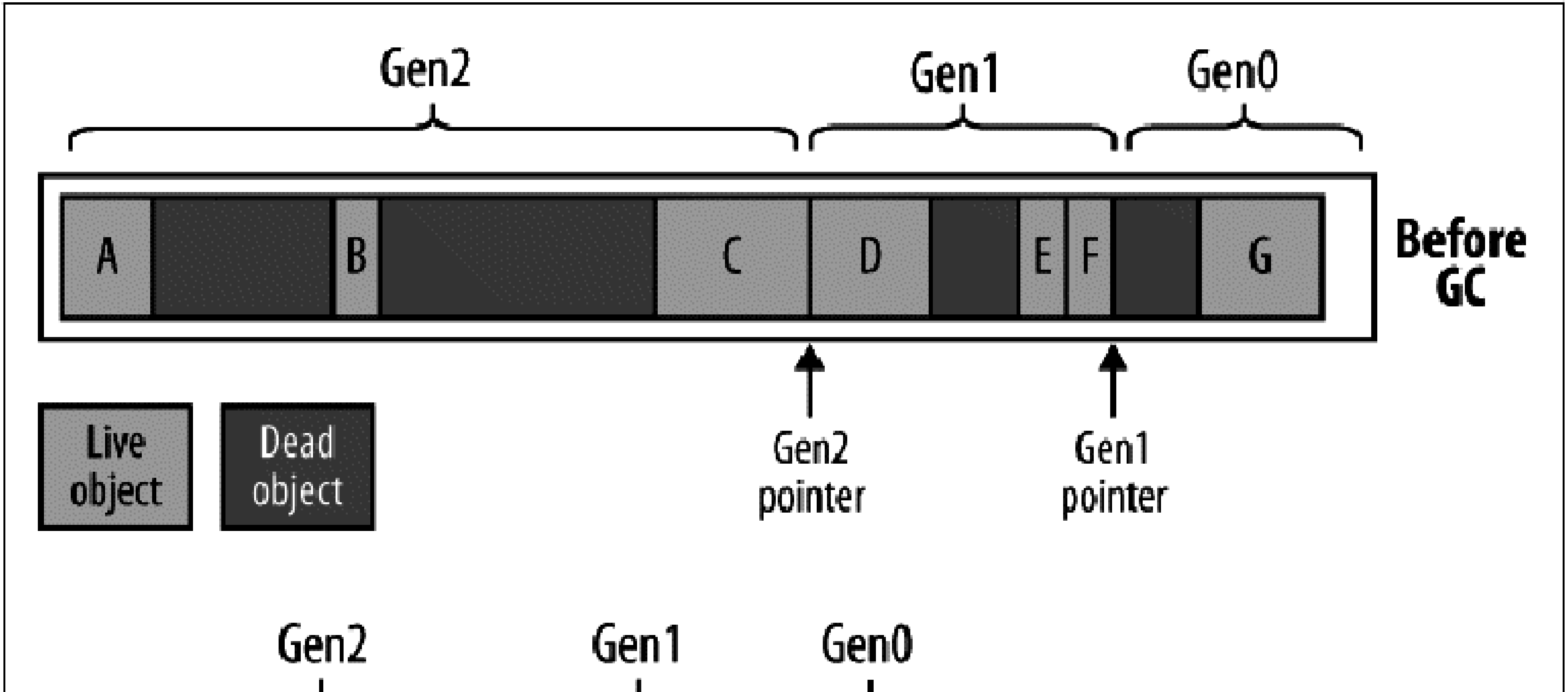
Basically, the GC divides the managed heap into three generations. Objects that have just been allocated are in *Gen0* and objects that have survived one collection cycle are in *Gen1*; all other objects are in *Gen2*.

The CLR keeps the Gen0 section relatively small (a maximum of 16 MB on the 32-bit workstation CLR, with a typical size of a few hundred KB to a few MB). When the Gen0 section fills up, the GC instigates a Gen0 collection—which happens relatively often. The GC applies a similar memory threshold to Gen1 (which acts as a buffer to Gen2), and so Gen1 collections are relatively quick and frequent too. Full atively often. The GC applies a similar memory threshold to Gen1 (which acts as a buffer to Gen2), and so Gen1 collections are relatively quick and frequent too. Full collections that include Gen2, however, take much longer and so happen infrequently. Figure 12-2 shows the effect of a full collection.

To give some very rough ballpark figures, a Gen0 collection might take less than 1 ms, which is not enough to be noticed in a typical application. A full collection, however, might take as long as 100 ms on a program with large object graphs. These figures depend on numerous factors and so may vary considerably—particularly in the case of Gen2 whose size is *unbounded* (unlike Gen0 and Gen1).

The upshot is that short-lived objects are very efficient in their use of the GC. The StringBuilders created in the following method would almost certainly be collected in a fast Gen0:

```
string Foo()
{
  var sb1 = new StringBuilder ("test");
  sb1.Append ("...");
  var sb2 = new StringBuilder ("test");
  var sb2 = new StringBuilder ("test");
  sb2.Append (sb1.ToString());
  return sb2.ToString();
}
```



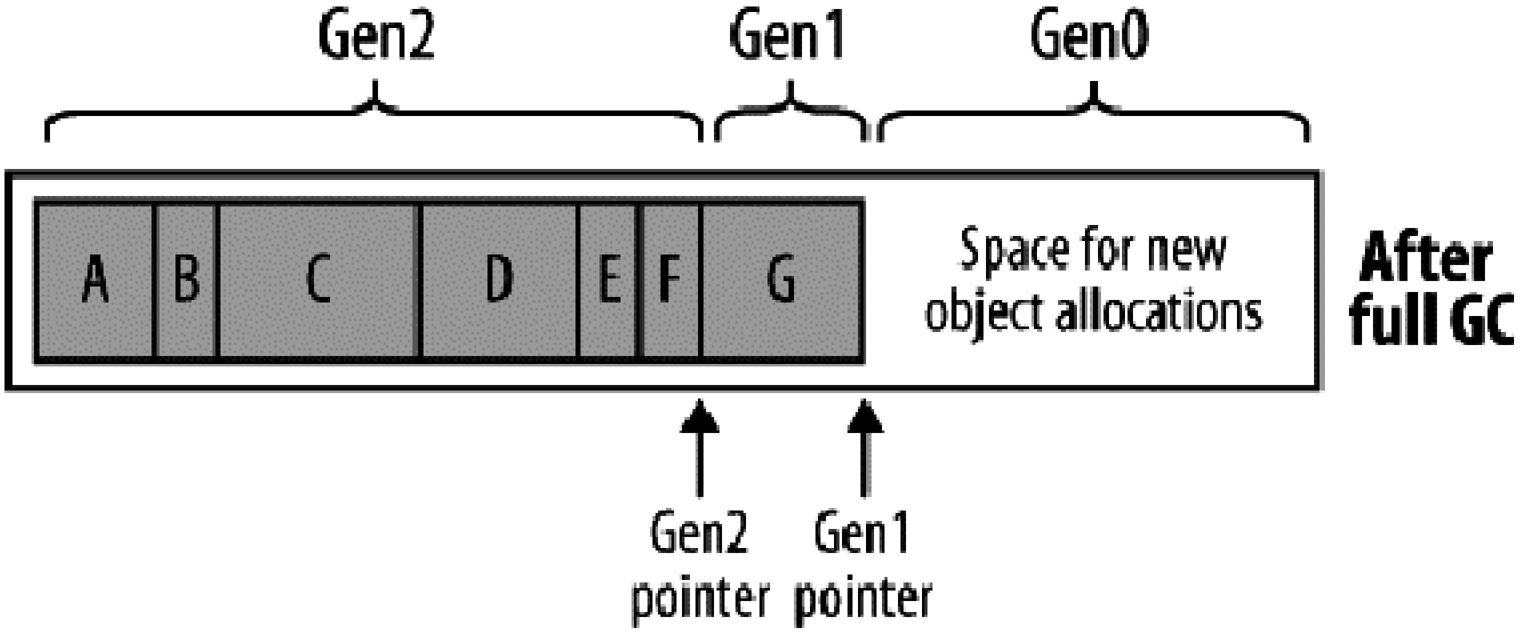


Figure 12-2. Heap generations

The large object heap

The GC uses a separate heap called the Large Object Heap (LOH) for objects larger than a certain threshold (currently 85,000 bytes). This avoids excessive Gen0 collections—without the LOH, allocating a series of 16 MB objects might trigger a Gen0 collection after every allocation.

The LOH is not subject to compaction, because moving large blocks of memory during garbage collection would be prohibitively expensive. This has two consequences:

●

●

Allocations can be slower, because the GC can't always simply allocate objects at the end of the heap—it must also look in the middle for gaps, and this requires maintaining a linked list of free memory blocks.†

The LOH is subject to *fragmentation*. This means that the freeing of an object maintaining a linked list of free memory blocks.[†]

The LOH is subject to *fragmentation*. This means that the freeing of an object can create a hole in the LOH that may be hard to fill later. For instance, a hole left by an 86,000-byte object can be filled only by an object of between 85,000 bytes and 86,000 bytes (unless adjoined by another hole).

The large object heap is also nongenerational: all objects are treated as Gen2.

[†] The same thing may occur occasionally in the generational heap due to pinning (see "The fixed Statement" on page 171 in Chapter 4).

Concurrent and background collection

The GC must freeze (block) your execution threads for periods during a collection. This includes the entire period during which a Gen0 or Gen1 collection takes place.

The GC makes a special attempt, though, at allowing threads to run during a Gen2 collection, as it's undesirable to freeze an application for a potentially long period. This optimization applies to the workstation version of the CLR only, which is used

This optimization applies to the workstation version of the CLR only, which is used on desktop versions of Windows (and on all versions of Windows with standalone applications). The rationale is that the latency from a blocking collection is less likely to be a problem for server applications thats don't have a user interface.

> A mitigating factor is that the server CLR leverages all available cores to perform GCs, so an eight-core server will perform a full GC many times faster. In effect, the server GC is tuned for throughput rather than latency.

The workstation optimization has historically been called *concurrent collection*. From CLR 4.0, it's been revamped and renamed to *background collection*. Background collection removes a limitation whereby a concurrent collection would cease to be concurrent if the Gen0 section filled up while a Gen2 collection was running. This means that from CLR 4.0, applications that continually allocate memory will be more responsive.

GC notifications (server CLR)

From Framework 3.5 SP1, the server version of the CLR can notify you just before

From Framework 3.5 SP1, the server version of the CLR can notify you just before a full GC will occur. This is intended for server farm configurations: the idea is that you divert requests to another server just before a collection. You then instigate the collection immediately and wait for it to complete before rerouting requests back to that server.

To start notification, call `GC.RegisterForFullGCNotification`. Then start up another thread (see Chapter 21) that first calls `GC.WaitForFullGCApproach`. When this method returns a `GCNotificationStatus` indicating that a collection is near, you can reroute requests to other servers and force a manual collection (see the following section). You then call `GC.WaitForFullGCComplete`: when this method returns, GC is complete and you can again accept requests. You then repeat the whole cycle.

Forcing Garbage Collection

You can manually force a GC at any time, by calling `GC.Collect`. Calling `GC.Collect` without an argument instigates a full collection. If you pass in an integer value, only generations to that value are collected, so `GC.Collect(0)` performs only a fast Gen0 collection.

a fast Gen0 collection.

In general, you get the best performance by allowing the GC to decide when to collect: forcing collection can hurt performance by unnecessarily promoting Gen0 objects to Gen1. It can also upset the GC's *self-tuning* ability, whereby the GC

dynamically tweaks the thresholds for each generation to maximize performance as the application executes.

There are exceptions, however. The most common case for intervention is when an application goes to sleep for a while: a good example is a Windows Service that performs a daily activity (checking for updates, perhaps). Such an application might use a `System.Timers.Timer` to initiate the activity every 24 hours. After completing the activity, no further code executes for 24 hours, which means that for this period, no memory allocations are made and so the GC has no opportunity to activate. Whatever memory the service consumed in performing its activity, it will continue to consume for the following 24 hours—even with an empty object graph! The solution is to call `GC.Collect` right after the daily activity completes.

To ensure the collection of objects for which collection is delayed by finalizers, you can take the additional step of calling `WaitForPendingFinalizers` and re-collecting:

To ensure the collection of objects for which collection is delayed by finalizers, you can take the additional step of calling `WaitForPendingFinalizers` and re-collecting:

```
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
```

Memory Pressure

Memory Pressure

The runtime decides when to initiate collections based on a number of factors, including the total memory load on the machine. If your program allocates unmanaged memory (Chapter 25), the runtime will get an unrealistically optimistic perception of its memory usage, because the CLR knows only about managed memory. You can mitigate this by telling the CLR to *assume* a specified quantity of unmanaged memory has been allocated, by calling `GC.AddMemoryPressure`. To undo this (when the unmanaged memory is released) call `GC.RemoveMemoryPressure`.

Managed Memory Leaks

In unmanaged languages such as C++, you must remember to manually deallocate memory when an object is no longer required; otherwise, a *memory leak* will result. In the managed world, this kind of error is impossible due to the CLR's automatic garbage collection system.

Nonetheless, large and complex .NET applications can exhibit a milder form of the same syndrome with the same end result: the application consumes more and more memory over its lifetime, until it eventually has to be restarted. The good news is memory over its lifetime, until it eventually has to be restarted. The good news is that managed memory leaks are usually easier to diagnose and prevent.

Managed memory leaks are caused by unused objects remaining alive by virtue of unused or forgotten references. A common candidate is event handlers—these hold a reference to the target object (unless the target is a static method). For instance, consider the following classes:

```
class Host
{
  public event EventHandler Click;
}
```

```
class Client
{
```

```
{
  Host _host;
  public Client (Host host)
  {
    _host = host;
    _host.Click += HostClicked;
  }

  void HostClicked (object sender, EventArgs e) { ... }
}
```

The following test class contains a method that instantiates 1,000 clients:

```
class Test
class Test
{
  static Host _host = new Host();

  public static void CreateClients()
  {
    Client[] clients = Enumerable.Range (0, 1000)
     .Select (i => new Client (_host))
     .ToArray();

    // Do something with clients ...
  }
}
```

You might expect that after CreateClients finishes executing, the 1,000 Client ob-

You might expect that after `CreateClients` finishes executing, the 1,000 `Client` objects will become eligible for collection. Unfortunately, each client has another referee: the `_host` object whose `Click` event now references each `Client` instance. This may go unnoticed if the `Click` event doesn't fire—or if the `HostClicked` method doesn't do anything to attract attention.

One way to solve this is to make `Client` implement `IDisposable`, and in the `Dispose` method, unhook the event handler:

```
public void Dispose() { _host.Click -= HostClicked; }
```

Consumers of `Client` then dispose of the instances when they're done with them:

```
Array.ForEach (clients, c => c.Dispose());
```

> In "Weak References" on page 494, we'll describe another solution to this problem, which can be useful in environments which tend not to use disposable objects (an example is WPF). In fact, the WPF framework offers a class called `WeakEventManager` that leverages a pattern employing weak references.

On the topic of WPF, *data binding* is another common cause for

On the topic of WPF, *data binding* is another common cause for memory leaks: the issue is described at *http://support.microsoft.com/kb/938416*.

Timers

Forgotten timers can also cause memory leaks (we discuss timers in Chapter 21). There are two distinct scenarios, depending on the kind of timer. Let's first look at the timer in the System.Timers namespace. In the following example, the Foo class (when instantiated) calls the tmr_Elapsed method once every second:

```
using System.Timers;

class Foo
{
```

```
{
  Timer _timer;
```

```
Foo()
{
  _timer = new System.Timers.Timer { Interval = 1000 };
  _timer.Elapsed += tmr_Elapsed;
  _timer.Start();
}
```

```
  void tmr_Elapsed (object sender, ElapsedEventArgs e) { ... }
}
```

Unfortunately, instances of Foo can never be garbage-collected! The problem is the .NET Framework itself holds references to active timers so that it can fire their Elapsed events. Hence:

- The .NET Framework will keep _timer alive.
- _timer will keep the Foo instance alive, via the tmr_Elapsed event handler.

The solution is obvious when you realize that Timer implements IDisposable. Disposing of the timer stops it and ensures that the .NET Framework no longer references the object:

posing of the timer stops it and ensures that the .NET Framework no longer references the object:

```
class Foo : IDisposable
{
  ...
  public void Dispose() { _timer.Dispose(); }
}
```

A good guideline is to implement `IDisposable` yourself if any field in your class is assigned an object that implements `IDisposable`.

The WPF and Windows Forms timers behave in exactly the same way, with respect to what's just been discussed.

The timer in the `System.Threading` namespace, however, is special. The .NET Framework doesn't hold references to active threading timers; it instead references the callback delegates directly. This means that if you forget to dispose of a threading callback delegates directly. This means that if you forget to dispose of a threading timer, a finalizer can (and will) fire—and this will automatically stop and dispose

the timer. This can create a different problem, however, which we can illustrate as follows:

```
static void Main()
{
  var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000);
  GC.Collect();
  System.Threading.Thread.Sleep (10000);    // Wait 10 seconds
}
```

```
static void TimerTick (object notUsed) { Console.WriteLine ("tick"); }
```

If this example is compiled in "release" mode (debugging disabled and optimizations enabled), the timer will be collected and finalized before it has a chance to fire even once! Again, we can fix this by disposing of the timer when we're done with it:

```
    using (var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000))
    {
```

```
using (var tmr = new System.Threading.Timer (TimerTick, null, 1000, 1000))
{
  GC.Collect();
  System.Threading.Thread.Sleep (10000);    // Wait 10 seconds
}
```

The implicit call to `tmr.Dispose` at the end of the `using` block ensures that the `tmr` variable is "used" and so not considered dead by the GC until the end of the block. Ironically, this call to `Dispose` actually keeps the object alive *longer*!

Diagnosing Memory Leaks

The easiest way to avoid managed memory leaks is to proactively monitor memory consumption as an application is written. You can obtain the current memory consumption of a program's objects as follows (the `true` argument tells the GC to perform a collection first):

```
long memoryUsed = GC.GetTotalMemory (true);
```

If you're practicing test-driven development, one possibility is to use unit tests to assert that memory is reclaimed as expected. If such an assertion fails, you then have to examine only the changes that you've made recently.

assert that memory is reclaimed as expected. If such an assertion fails, you then have to examine only the changes that you've made recently.

If you already have a large application with a managed memory leak, the *windbg.exe* tool can assist in finding it. There are also friendlier graphical tools such as Microsoft's CLR Profiler, SciTech's Memory Profiler, and Red Gate's ANTS Memory Profiler.

The CLR also exposes numerous Windows WMI counters to assist with resource monitoring.

Weak References

Occasionally, it's useful to hold a reference to an object that's "invisible" to the GC in terms of keeping the object alive. This is called a *weak reference*, and is implemented by the `System.WeakReference` class.

To use `WeakReference`, construct it with a target object as follows:

```
var sb = new StringBuilder ("this is a test");
var sb = new StringBuilder ("this is a test");
var weak = new WeakReference (sb);
Console.WriteLine (weak.Target);     // This is a test
```

If a target is referenced *only* by one or more weak references, the GC will consider the target eligible for collection. When the target gets collected, the Target property of the WeakReference will be null:

```
var weak = new WeakReference (new StringBuilder ("weak"));
Console.WriteLine (weak.Target);   // weak
GC.Collect();
Console.WriteLine (weak.Target);   // (nothing)
```

To avoid the target being collected in between testing for it being null and consuming it, assign the target to a local variable:

```
var weak = new WeakReference (new StringBuilder ("weak"));
var sb = (StringBuilder) weak.Target;
if (sb != null) { /* Do something with sb */ }
```

Once a target's been assigned to a local variable, it has a strong root and so cannot

Once a target's been assigned to a local variable, it has a strong root and so cannot be collected while that variable's in use.

Disposal and GC

The following class uses weak references to keep track of all `Widget` objects that have been instantiated, without preventing those objects from being collected:

```
class Widget
{
```

```
class Widget
{
  static List<WeakReference> _allWidgets = new List<WeakReference>();

  public readonly string Name;

  public Widget (string name)
  {
    Name = name;
    _allWidgets.Add (new WeakReference (this));
  }

  public static void ListAllWidgets()
  {
    foreach (WeakReference weak in _allWidgets)
    {
      Widget w = (Widget)weak.Target;
      if (w != null) Console.WriteLine (w.Name);
    }
```

```
        }
    }
 }
```

The only proviso with such a system is that the static list will grow over time, accumulating weak references with null targets. So you need to implement some cleanup strategy.

Weak References and Caching

One use for WeakReference is to cache large object graphs. This allows memory-intensive data to be cached briefly without causing excessive memory consumption:

```
_weakCache = new WeakReference (...);   // _weakCache is a field
...
var cache = _weakCache.Target;
if (cache == null) { /* Re-create cache & assign it to _weakCache */ }
```

```
var cache = _weakCache.Target;
if (cache == null) { /* Re-create cache & assign it to _weakCache */ }
```

This strategy may be only mildly effective in practice, because you have little control over when the GC fires and what generation it chooses to collect. In particular, if your cache remains in Gen0, it may be collected within microseconds (and remember that the GC doesn't collect only when memory is low—it collects regularly under normal memory conditions). So at a minimum, you should employ a two-level cache whereby you start out by holding strong references that you convert to weak references over time.

Weak References and Events

We saw earlier how events can cause managed memory leaks. The simplest solution is to either avoid subscribing in such conditions, or implement a `Dispose` method to unsubscribe. Weak references offer another solution.

Imagine a delegate that holds only weak references to its targets. Such a delegate would not keep its targets alive—unless those targets had independent referees. Of course, this wouldn't prevent a firing delegate from hitting an unreferenced target—in the time between the target being eligible for collection and the GC catching up with it. For such a solution to be effective, your code must be robust in that scenario in the time between the target being eligible for collection and the GC catching up with it. For such a solution to be effective, your code must be robust in that scenario. Assuming that is the case, a *weak delegate* class can be implemented as follows:

```
class WeakDelegate<TDelegate> where TDelegate : class
{
  List<WeakReference> _targets = new List<WeakReference>();

public WeakDelegate()
{
  if (!typeof (TDelegate).IsSubclassOf (typeof (Delegate)))
    throw new InvalidOperationException
     ("TDelegate must be a delegate type");
}

public void Combine (TDelegate target)
{
  if (target == null) return;
  foreach (Delegate d in (target as Delegate).GetInvocationList())
    _targets.Add (new WeakReference (d));
}
```

```
    _targets.Add (new WeakReference (d));
}

public void Remove (TDelegate target)
{
  if (target == null) return;
  foreach (Delegate d in (target as Delegate).GetInvocationList())
  {
    WeakReference weak = _targets.Find (w => d.Equals (w.Target));
    if (weak != null) _targets.Remove (weak);
  }
}
```

```
public TDelegate Target
{
  get
  {
    var deadRefs = new List<WeakReference>();
    {
      var deadRefs = new List<WeakReference>();
      Delegate combinedTarget = null;

foreach (WeakReference weak in _targets)
{
  Delegate target = (Delegate)weak.Target;
  if (target != null)
    combinedTarget = Delegate.Combine (combinedTarget, target);
  else
    deadRefs.Add (weak);
}
foreach (WeakReference weak in deadRefs)   // Remove dead references
  _targets.Remove (weak);                  // from _targets.

      return combinedTarget as TDelegate;
    }
    set
    {
```

```
    {
      _targets.Clear();
      Combine (value);
    }
  }
}
```

This code illustrates a number of interesting points in C# and the CLR. First, note that we check that TDelegate is a delegate type in the constructor. This is because of a limitation in C#—the following type constraint is illegal because C# considers System.Delegate a special type for which constraints are not supported:

```
    ... where TDelegate : Delegate   // Compiler doesn't allow this
```

Instead, we must choose a class constraint, and perform a runtime check in the constructor.

In the Combine and Remove methods, we perform the reference conversion from target to Delegate via the as operator, rather than the more usual cast operator. This is because C# disallows the cast operator with this type parameter—because of a potential ambiguity between a *custom conversion* and a *reference conversion*.

We then call GetInvocationList because these methods might be called with multicast delegates—delegates with more than one method recipient.

In the Target property, we build up a multicast delegate that combines all the delegates referenced by weak references whose targets are alive. We then clear out the

In the Target property, we build up a multicast delegate that combines all the delegates referenced by weak references whose targets are alive. We then clear out the remaining (dead) references from the list—to avoid the _targets list endlessly growing. (We could improve our class by doing the same in the Combine method; yet another improvement would be to add locks for thread safety [Chapter 21]).

The following illustrates how to consume this delegate in implementing an event:

```
public class Foo
{
  WeakDelegate<EventHandler> _click = new WeakDelegate<EventHandler>();

public event EventHandler Click
{
  add { _click.Combine (value); } remove { _click.Remove (value); }
}

  protected virtual void OnClick (EventArgs e)
  {
    {
      EventHandler target = _click.Target;
      if (target != null) target (this, e);
    }
  }
```

Notice that in firing the event, we assign `_click.Target` to a temporary variable before checking and invoking it. This avoids the possibility of targets being collected in the interim.

3

Diagnostics and Code Contracts

When things go wrong, it's important that information is available to aid in diagnosing the problem. An IDE or debugger can assist greatly to this effect—but it is usually available only during development. Once an application ships, the applicanosing the problem. An IDE or debugger can assist greatly to this effect—but it is usually available only during development. Once an application ships, the application itself must gather and record diagnostic information. To meet this requirement, the .NET Framework provides a set of facilities to log diagnostic information, monitor application behavior, detect runtime errors, and integrate with debugging tools if available.

From Framework 4.0, there are also a new set of types to enforce *code contracts*. These allow methods to interact through a set of mutual obligations, and fail *early* if those obligations are violated.

The types in this chapter are defined primarily in the `System.Diagnostics` and `System.Diagnostics.Contracts` namespaces.

Conditional Compilation

You can conditionally compile any section of code in C# with *preprocessor directives*. Preprocessor directives are special instructions to the compiler that begin with the # symbol (and, unlike other C# constructs, must appear on a line of their own). The preprocessor directives for conditional compilation are `#if`, `#else`, `#endif`, and `#elif`.

#elif.

The #if directive instructs the compiler to ignore a section of code unless a specified *symbol* has been defined. You can define a symbol with either the #define directive or a compilation switch. #define applies to a particular *file*; a compilation switch applies to a whole *assembly*:

```
#define TESTMODE            // #define directives must be at top of file
                            // Symbol names are uppercase by convention.
using System;
```

```
class Program
{
```

```
  static void Main()
  {
#if TESTMODE
#if TESTMODE
    Console.WriteLine ("in test mode!");
#endif
  }
}
```

```
// OUTPUT: in test mode!
```

If we deleted the first line, the program would compile with the `Console.WriteLine` statement completely eliminated from the executable.

The `#else` statement is analogous to C#'s `else` statement, and `#elif` is equivalent to `#else` followed by `#if`. The `||`, `&&`, and `!` operators can be used to perform *or*, *and*, and *not* operations:

```
#if TESTMODE && !PLAYMODE      // if TESTMODE and not PLAYMODE
  ...
```

Bear in mind, however, that you're not building an ordinary C# expression, and the symbols upon which you operate have absolutely no connection to *variables*—static

Bear in mind, however, that you're not building an ordinary C# expression, and the symbols upon which you operate have absolutely no connection to *variables*—static or otherwise.

To define a symbol assembly-wide, specify the `/define` switch when compiling:

```
csc Program.cs /define:TESTMODE,PLAYMODE
```

Visual Studio provides an option to enter conditional compilation symbols under Project Properties.

If you've defined a symbol at the assembly level and then want to "undefine" it for a particular file, you can do so with the `#undef` directive.

Conditional Compilation Versus Static Variable Flags

The preceding example could instead be implemented with a simple static field:

```
static internal bool TestMode = true;

static void Main()
{
```

```
static void Main()
{
  if (TestMode) Console.WriteLine ("in test mode!");
}
```

This has the advantage of allowing runtime configuration. So, why choose conditional compilation? The reason is that conditional compilation can take you places variable flags cannot, such as:

●

●

●

Conditionally including an attribute

Changing the declared type of variable

Switching between different namespaces or type aliases in a `using` directive—for example:

```
using TestType =
using TestType =
  #if V2
     MyCompany.Widgets.GadgetV2;
  #else
```

```
     MyCompany.Widgets.Gadget;
  #endif
```

You can even perform major refactoring under a conditional compilation directive, so you can instantly switch between old and new versions, and write libraries that can compile against multiple Framework versions, leveraging the latest Framework features where available.

Another advantage of conditional compilation is that debugging code can refer to types in assemblies that are not included in deployment.

types in assemblies that are not included in deployment.

The Conditional Attribute

The Conditional attribute instructs the compiler to ignore any calls to a particular class or method, if the specified symbol has not been defined.

To see how this is useful, suppose you write a method for logging status information as follows:

```
static void LogStatus (string msg)
{
  string logFilePath = ...
  System.IO.File.AppendAllText (logFilePath, msg + "\r\n");
}
```

Now imagine you wanted this to execute only if the LOGGINGMODE symbol is defined. The first solution is to wrap all calls to LogStatus around an #if directive:

```
#if LOGGINGMODE
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

```
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

This gives an ideal result, but it is tedious. The second solution is to put the #if directive inside the LogStatus method. This, however, is problematic should LogStatus be called as follows:

```
LogStatus ("Message Headers: " + GetComplexMessageHeaders());
```

GetComplexMessageHeaders would always get called—which might incur a perform-

GetComplexMessageHeaders would always get called—which might incur a performance hit.

We can combine the functionality of the first solution with the convenience of the second by attaching the Conditional attribute (defined in System.Diagnostics) to the LogStatus method:

```
[Conditional ("LOGGINGMODE")]
static void LogStatus (string msg)
{
  ...
}
```

This instructs the compiler to implicitly wrap any calls to LogStatus in an #if LOGGINGMODE directive. If the symbol is not defined, any calls to LogStatus get eliminated entirely in compilation—including their argument evaluation expressions. This works even if LogStatus and the caller are in different assemblies.

Another benefit of [Conditional] is that the conditionality check is performed when the *caller* is compiled, rather than

Another benefit of [Conditional] is that the conditionality check is performed when the *caller* is compiled, rather than when the *called method* is compiled. This is beneficial because it allows you to write a library containing methods such as `LogStatus`—and build just one version of that library.

The `Conditional` attribute is ignored at runtime—it's purely an instruction to the compiler.

Alternatives to the Conditional attribute

The `Conditional` attribute is useless if you need to dynamically enable or disable functionality at runtime: instead, you must use a variable-based approach. This leaves the question of how to elegantly circumvent the evaluation of arguments when calling conditional logging methods. A functional approach solves this:

```
using System;
using System.Linq;
```

```
class Program
{
  public static bool EnableLogging;
```

```
  static void LogStatus (Func<string> message)
  {
    string logFilePath = ...
    if (EnableLogging)
      System.IO.File.AppendAllText (logFilePath, message() + "\r\n");
  }
}
```

A lambda expression lets you call this method without syntax bloat:

```
    LogStatus ( () => "Message Headers: " + GetComplexMessageHeaders() );
```

If EnableLogging is false, GetComplexMessageHeaders is never evaluated.

Debug and Trace Classes

Debug and Trace Classes

Debug and Traceare static classes that provide basic logging and assertion capabilities. The two classes are very similar; the main differentiator is their intended use. The Debug class is intended for debug builds; the Trace class is intended for both debug and release builds. To this effect:

All methods of the Debug class are defined with [Conditional("DEBUG")].
All methods of the Trace class are defined with [Conditional("TRACE")].

This means that all calls that you make to Debug or Trace are eliminated by the compiler unless you define DEBUG or TRACE symbols. By default, Visual Studio defines both DEBUG and TRACE symbols in a project's *debug* configuration—and just the TRACE symbol in the *release* configuration.

Both the Debug and Trace classes provide Write, WriteLine, and WriteIf methods. By default, these send messages to the debugger's output window:

```
Debug.Write     ("Data");
Debug.Write     ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteIf   (x > y, "x is greater than y");
```

The Trace class also provides the methods TraceInformation, TraceWarning, and TraceError. The difference in behavior between these and the Write methods depends on the active TraceListeners (we will cover this in the section "TraceListener" on page 504).

Fail and Assert

The Debug and Trace classes both provide Fail and Assert methods. Fail sends the message to each TraceListener in the Debug or Trace class's Listeners collection (see the following section), which by default writes the message to the debug output as well as displaying it in a dialog:

```
Debug.Fail ("File data.txt does not exist!");
```

The dialog that appears asks you whether to ignore, abort, or retry. The latter then lets you attach a debugger, which is useful in instantly diagnosing the problem.

lets you attach a debugger, which is useful in instantly diagnosing the problem.

Assert simply calls Fail if the bool argument is false—this is called *making an assertion*. Specifying a failure message is optional:

```
    Debug.Assert (File.Exists ("data.txt"), "File data.txt does not exist!");
    var result = ...
    Debug.Assert (result != null);
```

The Write, Fail, and Assert methods are also overloaded to accept a string category

The Write, Fail, and Assert methods are also overloaded to accept a string category in addition to the message, which can be useful in processing the output.

An alternative way to make an assertion is to throw an exception if the opposite condition is true. This is a common practice when validating method arguments:

```
public void ShowMessage (string message)
{
  if (message != null) throw new ArgumentNullException ("message");
  ...
}
```

Assertions made in this way are compiled unconditionally and are less flexible in that you can't control the outcome of a failed assertion via TraceListeners.

We'll see soon how *code contracts* extend the principles of Fail and Assert, providing more power and flexibility.

TraceListener

The Debug and Trace classes each have a Listeners property, comprising a static collection of TraceListener instances. These are responsible for processing the content emitted by the Write, Fail, and Trace methods.

By default, the Listeners collection of each includes a single listener (DefaultTraceListener). The default listener has two key features:

-
-

When connected to a debugger such as Visual Studio, messages are written to the debug output window; otherwise, message content is ignored.

When the Fail method is called (or an assertion fails), a dialog appears asking the user whether to continue, abort, or retry (attach/debug)—regardless of whether a debugger is attached.

You can change this behavior by (optionally) removing the default listener, and then

You can change this behavior by (optionally) removing the default listener, and then adding one or more of your own. You can write trace listeners from scratch (by subclassing `TraceListener`) or use one of the predefined types:

●

●

●

●

`TextWriterTraceListener` writes to a `Stream` or `TextWriter` or appends to a file.

`EventLogTraceListener` writes to the Windows event log.

`EventProviderTraceListener` writes to the Event Tracing for Windows (ETW) subsystem in Windows Vista and later.

`WebPageTraceListener` writes to an ASP.NET web page.

`TextWriterTraceListener` is further subclassed to `ConsoleTraceListener`, `DelimitedListTraceListener`, `XmlWriterTraceListener`, and `EventSchemaTraceListener`.

None of these listeners display a dialog when `Fail` is called—

None of these listeners display a dialog when Fail is called—only DefaultTraceListener has this behavior.

The following example clears Trace's default listener, then adds three listeners—one that appends to a file, one that writes to the console, and one that writes to the Windows event log:

```
// Clear the default listener:
Trace.Listeners.Clear();
```

```
// Add a writer that appends to the trace.txt file:
Trace.Listeners.Add (new TextWriterTraceListener ("trace.txt"));
```

```
// Obtain the Console's output stream, then add that as a listener:
System.IO.TextWriter tw = Console.Out;
Trace.Listeners.Add (new TextWriterTraceListener (tw));
```

```
// Set up a Windows Event log source and then create/add listener.
// CreateEventSource requires administrative elevation, so this would
// typically be done in application setup.
if (!EventLog.SourceExists ("DemoApp"))
```

```
// typically be done in application setup.
if (!EventLog.SourceExists ("DemoApp"))
```

```
EventLog.CreateEventSource ("DemoApp", "Application");
```

```
Trace.Listeners.Add (new EventLogTraceListener ("DemoApp"));
```

In the case of the Windows event log, messages that you write with the Write, Fail, or Assert method always display as "Information" messages in the Windows event viewer. Messages that you write via the TraceWarning and TraceError methods, however, show up as warnings or errors.

TraceListener also has a Filter of type TraceFilter that you can set to control whether a message gets written to that listener. To do this, you either instantiate one of the predefined subclasses (EventTypeFilter or SourceFilter), or subclass TraceFilter and override the ShouldTrace method. You could use this to filter by category, for instance.

TraceListener also defines IndentLevel and IndentSize properties for controlling indentation, and the TraceOutputOptions property for writing extra data:

TraceListener also defines IndentLevel and IndentSize properties for controlling indentation, and the TraceOutputOptions property for writing extra data:

```
TextWriterTraceListener tl = new TextWriterTraceListener (Console.Out);
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

TraceOutputOptions are applied when using the Trace methods:

```
Trace.TraceWarning ("Orange alert");
```

```
DiagTest.vshost.exe Warning: 0 : Orange alert
    DateTime=2007-03-08T05:57:13.6250000Z
    Callstack=   at System.Environment.GetStackTrace(Exception e, Boolean
needFileInfo)
    at System.Environment.get_StackTrace()     at ...
```

Flushing and Closing Listeners

Some listeners, such as `TextWriterTraceListener`, ultimately write to a stream that is subject to caching. This has two implications:

●

●

A message may not appear in the output stream or file immediately.

You must close—or at least flush—the listener before your application ends; otherwise, you lose what's in the cache (up to 4 KB, by default, if you're writing to a file).

to a file).

The Trace and Debug classes provide static Close and Flush methods that call Close or Flush on all listeners (which in turn calls Close or Flush on any underlying writers and streams). Close implicitly calls Flush, closes file handles, and prevents further data from being written.

As a general rule, call Close before an application ends and call Flush anytime you want to ensure that current message data is written. This applies if you're using stream- or file-based listeners.

Trace and Debug also provide an AutoFlush property, which, if true, forces a Flush after every message.

It's a good policy to set `AutoFlush` to `true` on `Debug` and `Trace` if you're using any file- or stream-based listeners. Otherwise, if an unhandled exception or critical error occurs, the last 4 KB of diagnostic information may be lost.

Code Contracts Overview

We mentioned previously the concept of an *assertion*, whereby you check that certain conditions are met throughout your program. If a condition fails, it indicates a bug, which should typically be handled by invoking a debugger (in debug builds) or throwing an exception (in release builds).

Assertions follow the principle that if something goes wrong, it's best to fail early and close to the source of the error. This is usually better than trying to continue with invalid data—which can result in incorrect results or an exception later on in the program (which is harder to diagnose).

Historically, there have been two ways to enforce assertions:

Historically, there have been two ways to enforce assertions:

- By calling the `Assert` method on `Debug` or `Trace`
- By throwing exceptions (such as `ArgumentNullException`)

Framework 4.0 provides a new feature called *code contracts*, which replaces both of these approaches with a unified system. That system allows you to make not only simple assertions but also more powerful *contract*-based assertions.

Code contracts derive from the principle of "Design by Contract" from the Eiffel programming language, where functions interact with each other through a system of mutual obligations and benefits. Essentially, a function specifies *preconditions* that must be met by the client (caller), and in return guarantees *postconditions* which the client can depend on when the function returns.

The types for code contracts live in the `System.Diagnostics.Contracts` namespace.

Although the types that support code contracts are built into .NET Framework 4.0, the binary rewriter and the static checking tools are available as a separate download at the Microsoft DevLabs site. You must install these tools before you can use code contracts in Visual Studio 2010.

Why Use Code Contracts?

To illustrate, we'll write a method that adds an item to a list only if it's not already present—with two *preconditions* and a *postcondition*:

```
public static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // Precondition
  Contract.Requires (!list.IsReadOnly);      // Precondition
  Contract.Ensures (list.Contains (item));   // Postcondition
```

```
  if (list.Contains(item)) return false;
```

```
    if (list.Contains(item)) return false;
    list.Add (item);
    return true;
  }
```

The preconditions are defined by `Contract.Requires` and are verified when the method starts. The postcondition is defined by `Contract.Ensures` and is verified not where it appears in the code, but *when the method exits*.

Preconditions and postconditions act like assertions and, in this case, detect the following errors:

- Calling the method with a null or read-only list
- A bug in the method whereby we forgot to add the item to the list

Preconditions and postconditions must appear at the start of the method. This is conducive to good design: if you fail to fulfill the contract in subsequently writing the method, the error will be detected.

Moreover, these conditions form a discoverable `AddIfNotPresent` advertises to consumers:

contract

for that method.

●

●

“You must call me with a non-null writable list.”

“When I return, that list will contain the item you specified.”

when I return, that list will contain the item you specified.

These facts can be emitted into the assembly's XML documentation file (you can do this in Visual Studio by going to the Code Contracts tab of the Project Properties window, enabling the building of a contracts reference assembly, and checking "Emit Contracts into XML doc file"). Tools such as SandCastle can then incorporate contract details into documentation files.

Contracts also enable your program to be analyzed for correctness by static contract

Contracts also enable your program to be analyzed for correctness by static contract validation tools. If you try to call `AddIfNotPresent` with a `list` whose value might be null, for example, a static validation tool could warn you before you even run the program.

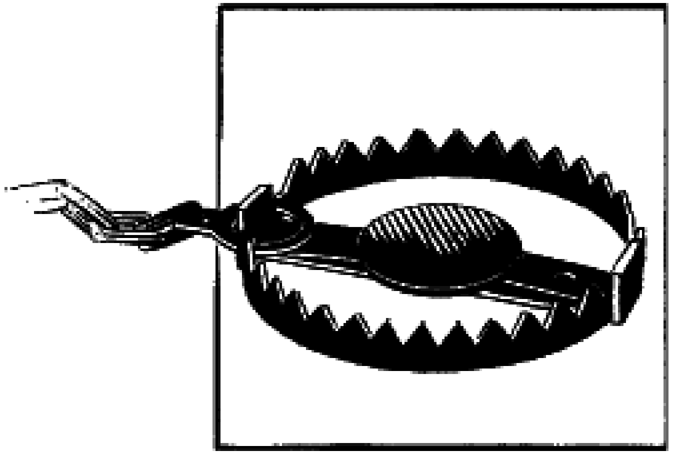
Another benefit of contracts is ease of use. In our example, it's easier to code the postcondition upfront than at both exit points. Contracts also support *object invariants*—which further reduce repetitive coding and make for more reliable enforcement.

Conditions can also be placed on interface members and abstract methods, something that is impossible with standard validation approaches. And conditions on virtual methods cannot be accidentally circumvented by subclasses.

Yet another benefit of code contracts is that contract violation behavior can be customized easily and in more ways than if you rely on calling `Debug.Assert` or throwing exceptions. And it's possible to ensure that contract violations are always recorded—even if contract violation exceptions are swallowed by exception handlers higher in the call stack.

The disadvantage of using code contracts is that the .NET implementation relies on a *binary rewriter*—a tool that mutates the assembly after compilation. This slows the build process, as well as complicating services that rely on calling the C# compiler (whether explicitly or via the `CSharpCodeProvider` class).

The enforcing of code contracts may also incur a runtime performance hit, although this is easily mitigated by scaling back contract checking in release builds.



Another limitation of code contracts is that you can't use them to enforce security-sensitive checks, because they can be circumvented at runtime (by handling the `ContractFailed` event).

Contract Principles

Code contracts comprise *preconditions*, *postconditions*, *assertions*, and *object invariants*. These are all discoverable assertions. They differ based on when they are verified:

●

●

●

●

Preconditions are verified when a function starts.

Postconditions are verified before a function exits.

Assertions are verified wherever they appear in the code.

Object invariants are verified after every public function in a class.

Code contracts are defined entirely by calling (static) methods in the `Contract` class. This makes contracts *language-independent*.

Contracts can appear not only in methods, but in other functions as well, such as constructors, properties, indexers, and operators.

Compilation

Almost all methods in the `Contract` class are defined with the `[Conditional("CONTRACTS_FULL")]` attribute. This means that unless you define the `CONTRACTS_FULL` symbol, (most) contract code is stripped out. Visual Studio defines the `CONTRACTS_FULL` symbol automatically if you enable contract checking in the Code Contracts tab of the Project Properties page. (For this tab to appear, you must download and install the Contracts tools from the Microsoft DevLabs site.)

Removing the `CONTRACTS_FULL` symbol might seem like an easy way to disable all contract checking. However, it doesn't apply to `Requires<TException>` conditions (which we'll describe in detail soon).

The only way to disable contracts in code that uses `Requires<TException>` is to enable the `CONTRACTS_FULL` symbol and then get the binary rewriter to strip out contract code by choosing an enforcement level of "none".

The binary rewriter

After compiling code that contains contracts, you must call the binary rewriter tool, *ccrewrite.exe* (Visual Studio does this automatically if contract checking is enabled). The binary rewriter moves postconditions (and object invariants) into the right *ccrewrite.exe* (Visual Studio does this automatically if contract checking is enabled). The binary rewriter moves postconditions (and object invariants) into the right place, calls any conditions and object invariants in overridden methods, and replaces calls to `Contract` with calls to a *contracts runtime class*. Here's a (simplified) version of what our earlier example would look like after rewriting:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  __ContractsRuntime.Requires (list != null);
  __ContractsRuntime.Requires (!list.IsReadOnly);
  bool result;
  if (list.Contains (item))
    result = false;
  else
  {
    list.Add (item);
    result = true;
  }
  __ContractsRuntime.Ensures (list.Contains (item));   // Postcondition
  return result;
}
```

If you fail to call the binary rewriter, `Contract` won't get replaced `__ContractsRuntime` and the former will end up throwing exceptions.

with

The __ContractsRuntime type is the default contracts runtime class. In advanced scenarios, you can specify your own contracts runtime class via the /rw switch or Visual Studio's Code Contracts tab in Project Properties runtime class via the /rw switch or Visual Studio's Code Contracts tab in Project Properties.

Because __ContractsRuntime is shipped with the binary rewriter (which is not a standard part of the .NET Framework), the binary rewriter actually injects the __ContractsRuntime class into your compiled assembly. You can examine its code by disassembling any assembly that enables code contracts.

The binary rewriter also offers switches to strip away some or all contract checking: we describe these in "Selectively Enforcing Contracts" on page 521. You typically enable full contract checking in debug build configurations and a subset of contract checking in release configurations.

Asserting versus throwing on failure

The binary rewriter also lets you choose between displaying a dialog and throwing a ContractException upon contract failure. The former is typically used for debug builds; the latter for release builds. To enable the latter, specify /throwonfailure when calling the binary rewriter, or uncheck the "Assert on contract failure" checkbox in Visual Studio's Code Contracts tab in Project Properties.

when calling the binary rewriter, or uncheck the "Assert on contract failure" checkbox in Visual Studio's Code Contracts tab in Project Properties.

See "Dealing with Contract Failure" on page 519 for more detail.

Purity

All functions that you call from arguments passed to contract methods (`Requires`, `Assumes`, `Assert`, etc.) must be *pure*—that is, side-effect-free (they must not alter the values of fields). You must signal to the binary rewriter that any functions you call are pure by applying the `[Pure]` attribute:

```
[Pure]
public static bool IsValidUri (string uri) { ... }
```

This makes the following legal:

```
Contract.Requires (IsValidUri (uri));
```

The contract tools implicitly assume that all property get accessors are pure, as are all C# operators (+, *, %, etc.) and members on selected Framework types, including string, Contract, Type, System.IO.Path, and LINQ's query operators. It also assumes that methods invoked via delegates marked with the [Pure] attribute are pure (the Comparison<T> and Predicate<T> attributes are marked with this attribute).

Preconditions

You can define code contract preconditions by calling Contract.Requires, Contract.Requires<TException>, or Contract.EndContractBlock.

Contract.Requires

Calling Contract.Requires at the start of a function enforces a precondition:

```
static string ToProperCase (string s)
{
  Contract.Requires (!string.IsNullOrEmpty(s));
```

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

This is like making an assertion, except that the precondition forms a discoverable fact about your function that can be extracted from the compiled code and consumed by documentation or static checking tools (so that they can warn you should they see some code elsewhere in your program that tries to call `ToProperCase` with a null or empty string).

A further benefit of preconditions is that subclasses that override virtual methods with preconditions cannot prevent the base class method's preconditions from being checked. And preconditions defined on *interface* members will be implicitly woven into the concrete implementations (see "Contracts on Interfaces and Abstract Methods" on page 518).

Preconditions should access only members that are at least as accessible as the function itself—this ensures that callers can make sense of the contract. If you need to read or call less accessible members, it's likely that you're validating *internal state* rather than enforcing the *calling contract*, in which case you should make an assertion instead.

You can call `Contract.Requires` as many times as necessary at the start of the method to enforce different conditions.

What Should You Put in Preconditions?

The guideline from the Code Contracts team is that preconditions should:

- Be possible for the client (caller) to easily validate

- Be possible for the client (caller) to easily validate
- Rely only on data & functions at least as accessible as the method itself
- Always indicate a *bug* if violated

A consequence of the last point is that a client should never specifically “catch” a contract failure (the `ContractException` type, in fact, is internal to help enforce that principle). Instead, the client should call the target properly; if it fails, this indicates a bug that should be handled via your general exception backstop (which may include terminating the application). In other words, if you decide control-flow or do other things based on a precondition failure, it’s not really a contract because you can continue executing if it fails.

This leads to the following advice, when choosing between preconditions and throwing ordinary exceptions:

- If failure *always* indicates a bug in the client, favor a precondition.
- If failure indicates an *abnormal condition*, which *may* mean a bug in the client, throw a (catchable) exception instead.

To illustrate, suppose we’re writing the `Int32.Parse` function. It’s reasonable to assume that a null input string always indicates a bug in the caller, so we’d enforce this with a precondition:

```
public static int Parse (string s)
{
```

```
public static int Parse (string s)
{
  Contract.Requires (s != null);
}
```

Next, we need to check that the string contains only digits and symbols such as + and - (in the right place). It would place an unreasonable burden on the caller to validate this and so we'd enforce it not as a precondition, but a manual check that throws a (catchable) FormatException if violated.

To illustrate the member accessibility issue, consider the following code, which often appears in types implementing the IDisposable interface:

```
public void Foo()
{
  if (_isDisposed)  // _isDisposed is a private field
    throw new ObjectDisposedException();
  ...
}
```

This check should not be made into a precondition unless we make _isDisposed accessible to the caller (by refactoring it into a publicly readable property, for instance).

Finally, consider the File.ReadAllText method. The following would be *inappropriate* use of a precondition:

```
public static string ReadAllText (string path)
{
    Contract.Requires (File.Exists (path));
    ...
}
```

```
    }
```

The caller cannot reliably know that the file exists before calling this method (it could be deleted between making that check and calling the method). So, we'd enforce this in the old-fashioned way—by throwing a catchable `FileNotFoundException` instead.

Contract.Requires<TException>

The introduction of code contracts challenges the following deeply entrenched pattern established in the .NET Framework from version 1.0:

```
static void SetProgress (string message, int percent)
{
  if (message == null)
    throw new ArgumentNullException ("message");
  if (percent < 0 || percent > 100)
    throw new ArgumentOutOfRangeException ("percent");
    throw new ArgumentOutOfRangeException ("percent");
  ...
}
static void SetProgress (string message, int percent)
{
  Contract.Requires (message != null);
  Contract.Requires (percent >= 0 && percent <= 100);
  ...
}
```

```
 // Classic approach
;
 // Modern approach
```

If you have a large assembly that enforces classic argument checking, writing new methods with preconditions will create an inconsistent library: some methods will throw argument exceptions whereas others will throw a `ContractException`. One methods with preconditions will create an inconsistent library. Some methods will throw argument exceptions whereas others will throw a ContractException. One solution is to update all existing methods to use contracts, but this has two problems:

●

●

It's time-consuming.

Callers may have come to *depend* on an exception type such as ArgumentNullException being thrown. (This almost certainly indicates bad design, but may be the reality nonetheless.)

The solution is to call the generic version of Contract.Requires. This lets you specify an exception type to throw upon failure:

```
Contract.Requires<ArgumentNullException> (message != null, "message");
Contract.Requires<ArgumentOutOfRangeException>
 (percent >= 0 && percent <= 100, "percent");
```

(The second argument gets passed to the constructor of the exception class).

This results in the same behavior as with old-fashioned argument checking, while

(The second argument gets passed to the constructor of the exception class).

This results in the same behavior as with old-fashioned argument checking, while delivering the benefits of contracts (conciseness, support for interfaces, implicit documentation, static checking, and runtime customization).

The specified exception is thrown only if you specify `/throwonfailure` when rewriting the assembly (or *uncheck* the *Assert on Contract Failure* checkbox in Visual Studio). Otherwise, a dialog appears.

It's also possible to specify a contract-checking level of *ReleaseRequires* in the binary rewriter (see "Selectively Enforcing Contracts" on page 521). Calls to the generic `Contract.Requires<TException>` then remain in place while all other checks are stripped away: this results in an assembly that behaves just as in the past.

Contract.EndContractBlock

The `Contract.EndContractBlock` method lets you get the benefit of code contracts with traditional argument-checking code—avoiding the need to refactor code that you wrote prior to Framework 4.0. All you do is call this method after performing manual argument checks:

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.EndContractBlock();
   • • •
  ...
}
```

The binary rewriter then converts this code into something equivalent to:

```
static void Foo (string name)
{
  Contract.Requires<ArgumentNullException> (name != null, "name");
  ...
}
```

The code that precedes EndContractBlock must comprise simple statements of the form:

```
if <condition> throw <expression>;
```

You can mix traditional argument checking with code contract calls: simply put the latter after the former:

```
static void Foo (string name)
```

latter after the former.

```
static void Foo (string name)
{
  if (name == null) throw new ArgumentNullException ("name");
  Contract.Requires (name.Length >= 2);
  ...
}
```

Calling any of the contract-enforcing methods implicitly ends the contract block.

The point is to define a region at the beginning of the method where the contract rewriter knows that every `if` statement is part of a contract. Calling any of the contract-enforcing methods implicitly extends the contract block, so you don't need to use `EndContractBlock` if you use another method such as `Contract.Ensures`, etc.

Preconditions and Overridden Methods

An overridden method cannot add preconditions, because doing so would *change the contract* (by making it more restrictive)—breaking the principles of polymorphism.

polymorphism.

(Technically, the designers could have allowed overridden methods to *weaken* preconditions; they decided against this because the scenarios weren't sufficiently compelling to justify adding this complexity).

The binary rewriter ensures that a base method's preconditions are always enforced in subclasses—whether or not the overridden method calls the base method.

Postconditions

Contract.Ensures

`Contract.Ensures` enforces a postcondition: something which must be true when the method exits. We saw an example earlier:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // Precondition
```

```
    {
      Contract.Requires (list != null);          // Precondition
      Contract.Ensures (list.Contains (item));   // Postcondition
      if (list.Contains(item)) return false;
      list.Add (item);
      return true;
    }
```

The binary rewriter moves postconditions to the exit points of the method. Postconditions are checked if you return early from a method (as in this example)—but not if you return early via an unhandled exception.

Unlike preconditions, which detect misuse by the *caller*, postconditions detect an error in the function itself (rather like assertions). Therefore, postconditions may access private state (subject to the caveat stated shortly, in "Postconditions and Overridden Methods" on page 516).

Postconditions and Thread Safety

Multithreaded scenarios (Chapter 21) challenge the usefulness of postconditions. For instance, suppose we wrote a thread-safe wrapper for a List<T> with a method as follows:

For instance, suppose we wrote a thread-safe wrapper for a `List<T>` with a method as follows:

```
public class ThreadSafeList<T>
{
  List<T> _list = new List<T>();
  object _locker = new object();

  public bool AddIfNotPresent (T item)
  {
    Contract.Ensures (_list.Contains (item));
    lock (_locker)
    {
      if (_list.Contains(item)) return false;
      _list.Add (item);
      return true;
    }
  }

  public void Remove (T item)
  {
    lock (_locker)
      _list.Remove (item);
  }
}
```

The postcondition in the `AddIfNotPresent` method is checked *after* the lock is

The postcondition in the AddIfNotPresent method is checked *after* the lock is released—at which point the item may no longer exist in the list if another thread called Remove right then. There is currently no workaround for this problem, other than to enforce such conditions as assertions (see next section) rather than postconditions.

Contract EnsuresOnThrow<TException>

Contract.EnsuresOnThrow<TException>

Occasionally, it's useful to ensure that a certain condition holds should a particular type of exception be thrown. The EnsuresOnThrow method does exactly this:

```
Contract.EnsuresOnThrow<WebException> (this.ErrorMessage != null);
```

Contract.Result<T> and Contract.ValueAtReturn<T>

Because postconditions are not evaluated until a function ends, it's reasonable to want to access the return value of a method. The Contract.Result<T> method does exactly that:

```
Random _random = new Random();
int GetOddRandomNumber()
{
  Contract.Ensures (Contract.Result<int>() % 2 == 1);
  return _random.Next (100) * 2 + 1;
}
```

The Contract.ValueAtReturn<T> method fulfills the same function—but for ref and

The `Contract.ValueAtReturn<T>` method fulfills the same function—but for `ref` and `out` parameters.

Contract.OldValue<T>

`Contract.OldValue<T>` returns the original value of a method parameter. This is useful with postconditions because the latter are checked at the *end* of a function. Therefore, any expressions in postconditions that incorporate parameters will read the *modified* parameter values.

For example, the postcondition in the following method will always fail:

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length < s.Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

Here's how we can correct it:

Here’s how we can correct it:

```
static string Middle (string s)
{
  Contract.Requires (s != null && s.Length >= 2);
  Contract.Ensures (Contract.Result<string>().Length <
                    Contract.OldValue (s).Length);
  s = s.Substring (1, s.Length - 2);
  return s.Trim();
}
```

Postconditions and Overridden Methods

An overridden method cannot circumvent postconditions defined by its base, but it can add new ones. The binary rewriter ensures that a base method’s postconditions are always checked—even if the overridden method doesn’t call the base implementation.

For the reason just stated, postconditions on virtual methods should not access private members. Doing so will result in the binary rewriter weaving code into the subclass that will try to access private members in the base class—causing a runtime error.

Assertions and Object Invariants

In addition to preconditions and postconditions, the code contracts API lets you