Chapter 12. Disposal and Garbage Collection

Some objects require explicit tear-down code to release resources such as open files, locks, operating system handles, and unmanaged objects. In .NET parlance, this is called *disposal*, and it is supported through the `IDisposable` interface. The managed memory occupied by unused objects must also be reclaimed at some point; this function is known as *garbage collection* and is performed by the CLR.

Disposal differs from garbage collection in that disposal is usually explicitly instigated; garbage collection is totally automatic. In other words, the programmer takes care of such things as releasing file handles, locks, and operating system resources, while the CLR takes care of releasing memory.

This chapter discusses both disposal and garbage collection, and also describes C# finalizers and the pattern by which they can provide a backup for disposal. Lastly, we discuss the intricacies of the garbage collector and other memory management options.

IDisposable, Dispose, and Close

.NET defines a special interface for types requiring a tear-down method:

```
public interface IDisposable
{
  void Dispose();
}
```

C#'s `using` statement provides a syntactic shortcut for calling `Dispose` on objects that implement `IDisposable`, using a `try`/`finally` block:

```
using (FileStream fs = new FileStream ("myFile.txt", FileMode.Open))
{
  // ... Write to the file ...
}
```

The compiler converts this to the following:

```
FileStream fs = new FileStream ("myFile.txt", FileMode.Open);
try
{
  // ... Write to the file ...
}
finally
{
  if (fs != null) ((IDisposable)fs).Dispose();
}
```

The `finally` block ensures that the `Dispose` method is called even when an exception is thrown or the code exits the block early.

Similarly, the following syntax ensures disposal as soon as `fs` goes out of scope:

```
using FileStream fs = new FileStream ("myFile.txt", FileMode.Open);

// ... Write to the file ...
```

In simple scenarios, writing your own disposable type is just a matter of implementing `IDisposable` and writing the `Dispose` method:

```
sealed class Demo : IDisposable
{
  public void Dispose()
  {
    // Perform cleanup / tear-down.
    ...
  }
}
```

> **NOTE**
>
> This pattern works well in simple cases and is appropriate for sealed classes. In "Calling Dispose from a Finalizer", we describe a more elaborate pattern that can provide a backup for consumers that forget to call `Dispose`. With unsealed types, there's a strong case for following this latter pattern from the outset—otherwise, it becomes very messy if the subtype wants to add such functionality itself.

Standard Disposal Semantics

.NET follows a de facto set of rules in its disposal logic. These rules are not hardwired to .NET or the C# language in any way; their purpose is to define a consistent protocol to consumers. Here they are:

1. After an object has been disposed, it's beyond redemption. It cannot be reactivated, and calling its methods or properties (other than `Dispose`) throws an `ObjectDisposedException`.
2. Calling an object's `Dispose` method repeatedly causes no error.
3. If disposable object *x* "owns" disposable object *y*, *x*'s `Dispose` method automatically calls *y*'s `Dispose` method—unless instructed otherwise.

These rules are also helpful when writing your own types, though they're not mandatory. Nothing prevents you from writing an "Undispose" method other than, perhaps, the flak you might cop from colleagues!

According to rule 3, a container object automatically disposes its child objects. A good example is a Windows Forms container control such as a `Form` or `Panel`. The container can host many child controls, yet you don't dispose every one of them explicitly; closing or disposing the parent control or form takes care of the whole lot. Another example is when you wrap a `FileStream` in a `DeflateStream`. Disposing the `DeflateStream` also disposes the `FileStream`—unless you instructed otherwise in the constructor.

Close and Stop

Some types define a method called `Close` in addition to `Dispose`. The .NET BCL is not completely consistent on the semantics of a `Close` method, although in nearly all cases it's either of the following:

- Functionally identical to `Dispose`
- A functional *subset* of `Dispose`

An example of the latter is `IDbConnection`: a `Closed` connection can be re-`Open`ed; a `Dispose`d connection cannot. Another example is a Windows `Form` activated with `ShowDialog`: `Close` hides it; `Dispose` releases its resources.

Some classes define a `Stop` method (e.g., `Timer` or `HttpListener`). A `Stop` method may release unmanaged resources, like `Dispose`, but unlike `Dispose`, it allows for re-`Start`ing.

When to Dispose

A safe rule to follow (in nearly all cases) is "if in doubt, dispose." Objects wrapping an unmanaged resource handle will nearly always require disposal in order to free the handle. Examples include file or network streams, network sockets, Windows Forms controls, GDI+ pens, brushes, and bitmaps. Conversely, if a type is disposable, it will often (but not always) reference an unmanaged handle, directly or indirectly. This is because unmanaged handles provide the gateway to the "outside world" of OS resources, network connections, and database locks—the primary means by which objects can create trouble outside of themselves if improperly abandoned.

There are, however, three scenarios for *not* disposing:

- When you don't "own" the object—for example, when obtaining a *shared* object via a static field or property
- When an object's `Dispose` method does something that you don't want

- When an object's `Dispose` method is unnecessary *by design*, and disposing that object would add complexity to your program

The first category is rare. The main cases are in the `System.Drawing` namespace: the GDI+ objects obtained through *static fields or properties* (such as `Brushes.Blue`) must never be disposed because the same instance is used throughout the life of the application. Instances that you obtain through constructors, however (such as `new SolidBrush`), *should* be disposed, as should instances obtained through static *methods* (such as `Font.FromHdc`).

The second category is more common. There are some good examples in the `System.IO` and `System.Data` namespaces:

Type	Disposal function	When not to dispose
`MemoryStream`	Prevents further I/O	When you later need to read/write the stream
`StreamReader`, `StreamWriter`	Flushes the reader/writer and closes the underlying stream	When you want to keep the underlying stream open (you must then call `Flush` on a `StreamWriter` when you're done)
`IDbConnection`	Releases a database connection and clears the connection string	If you need to re-`Open` it, you should call `Close` instead of `Dispose`
`DbContext` (EF Core)	Prevents further use	When you might have lazily evaluated queries connected to that context

`MemoryStream`'s `Dispose` method disables only the object; it doesn't perform any critical cleanup because a `MemoryStream` holds no unmanaged handles or other such resources.

The third category includes the classes such as `StringReader` and `StringWriter`. These types are disposable under the duress of their base class rather than through a genuine need to perform essential cleanup. If you happen to instantiate and work with such an object entirely in one method, wrapping it in a `using` block adds little inconvenience. But if the object is longer lasting, keeping track of when it's no longer used so that you can dispose of it adds unnecessary complexity. In such cases, you can simply ignore object disposal.

> **NOTE**
>
> Ignoring disposal can sometimes incur a performance cost (see "Calling Dispose from a Finalizer").

Clearing Fields in Disposal

In general, you don't need to clear an object's fields in its `Dispose` method. However, it is good practice to unsubscribe from events that the object has subscribed to internally over its lifetime (for an example, see "Managed Memory Leaks"). Unsubscribing from such events prevents receiving unwanted event notifications—and prevents unintentionally keeping the object alive in the eyes of the garbage collector (GC).

> **NOTE**
>
> A `Dispose` method itself does not cause (managed) memory to be released—this can happen only in garbage collection.

It's also worth setting a field to indicate that the object is disposed so that you can throw an `ObjectDisposedException` if a consumer later tries to call members on the object. A good pattern is to use a publicly readable automatic property for this:

```
public bool IsDisposed { get; private set; }
```

Although technically unnecessary, it can also be good to clear an object's own event handlers (by setting them to `null`) in the `Dispose` method. This eliminates the possibility of those events firing during or after disposal.

Occasionally, an object holds high-value secrets, such as encryption keys. In these cases, it can make sense to clear such data from fields during disposal (to avoid potential discovery by other processes on the machine when the memory is later released to the operating system). The `SymmetricAlgorithm` class in `System.Security.Cryptography` does exactly this by calling `Array.Clear` on the byte array holding the encryption key.

Anonymous Disposal

Sometimes, it's useful to implement `IDisposable` without having to write a class. For instance, suppose that you want to expose methods on a class that suspend and resume event processing:

```
class Foo
{
  int _suspendCount;

  public void SuspendEvents() => _suspendCount++;
  public void ResumeEvents() => _suspendCount--;

  void FireSomeEvent()
  {
    if (_suspendCount == 0)
      ... fire some event ...
  }
```

```
  ...
}
```

Such an API is clumsy to use. Consumers must remember to call `ResumeEvents`. And to be robust, they must do so in a `finally` block (in case an exception is thrown):

```
var foo = new Foo();
foo.SuspendEvents();
try
{
  ... do stuff ...      // Because an exception could be thrown here
}
finally
{
  foo.ResumeEvents();   // ...we must call this in a finally block
}
```

A better pattern is to do away with `ResumeEvents` and have `SuspendEvents` return an `IDisposable`. Consumers can then do this:

```
using (foo.SuspendEvents())
{
  ... do stuff ...
}
```

The problem is that this pushes work onto whoever has to implement the `SuspendEvents` method. Even with a good effort to reduce whitespace, we end up with this extra clutter:

```
public IDisposable SuspendEvents()
{
  _suspendCount++;
  return new SuspendToken (this);
}

class SuspendToken : IDisposable
{
  Foo _foo;
```

```
  public SuspendToken (Foo foo) => _foo = foo;
  public void Dispose()
  {
    if (_foo != null) _foo._suspendCount--;
    _foo = null;  // Prevent against consumer disposing twice
  }
}
```

The *anonymous disposal* pattern solves this problem. With the following reusable class:

```
public class Disposable : IDisposable
{
  public static Disposable Create (Action onDispose)
    => new Disposable (onDispose);

  Action _onDispose;
  Disposable (Action onDispose) => _onDispose = onDispose;

  public void Dispose()
  {
    _onDispose?.Invoke();   // Execute disposal action if non-null.
    _onDispose = null;      // Ensure it can't execute a second time.
  }
}
```

we can reduce our `SuspendEvents` method to the following:

```
public IDisposable SuspendEvents()
{
  _suspendCount++;
  return Disposable.Create (() => _suspendCount--);
}
```

Automatic Garbage Collection

Regardless of whether an object requires a `Dispose` method for custom tear-down logic, at some point the memory it occupies on the heap must be freed. The CLR handles this side of it entirely automatically via an

automatic GC. You never deallocate managed memory yourself. For example, consider the following method:

```
public void Test()
{
  byte[] myArray = new byte[1000];
  ...
}
```

When `Test` executes, an array to hold 1,000 bytes is allocated on the memory heap. The array is referenced by the variable `myArray`, stored on the local variable stack. When the method exits, this local variable `myArray` pops out of scope, meaning that nothing is left to reference the array on the memory heap. The orphaned array then becomes eligible to be reclaimed in garbage collection.

> **NOTE**
>
> In debug mode with optimizations disabled, the lifetime of an object referenced by a local variable extends to the end of the code block to ease debugging. Otherwise, it becomes eligible for collection at the earliest point at which it's no longer used.

Garbage collection does not happen immediately after an object is orphaned. Rather like garbage collection on the street, it happens periodically, although (unlike garbage collection on the street) not to a fixed schedule. The CLR bases its decision on when to collect upon a number of factors, such as the available memory, the amount of memory allocation, and the time since the last collection (the GC self-tunes to optimize for an application's specific memory access patterns). This means that there's an indeterminate delay between an object being orphaned and being released from memory. This delay can range from nanoseconds to days.

> **NOTE**
>
> The GC doesn't collect all garbage with every collection. Instead, the memory manager divides objects into *generations*, and the GC collects new generations (recently allocated objects) more frequently than old generations (long-lived objects). We discuss this in more detail in "How the GC Works".

GARBAGE COLLECTION AND MEMORY CONSUMPTION

The GC tries to strike a balance between the time it spends doing garbage collection and the application's memory consumption (working set). Consequently, applications can consume more memory than they need, particularly if large temporary arrays are constructed.

You can monitor a process's memory consumption via the Windows Task Manager or Resource Monitor—or programmatically by querying a performance counter:

```
// These types are in System.Diagnostics:
string procName = Process.GetCurrentProcess().ProcessName;
using PerformanceCounter pc = new PerformanceCounter
      ("Process", "Private Bytes", procName);
Console.WriteLine (pc.NextValue());
```

This queries the *private working set*, which gives the best overall indication of your program's memory consumption. Specifically, it excludes memory that the CLR has internally deallocated and is willing to rescind to the OS should another process need it.

Roots

A *root* is something that keeps an object alive. If an object is not directly or indirectly referenced by a root, it will be eligible for garbage collection.

A root is one of the following:

- A local variable or parameter in an executing method (or in any method in its call stack)
- A static variable
- An object on the queue that stores objects ready for finalization (see the next section)

It's impossible for code to execute in a deleted object, so if there's any possibility of an (instance) method executing, its object must somehow be referenced in one of these ways.

Note that a group of objects that reference one another cyclically are considered dead without a root referee (see Figure 12-1). To put it in another way, objects that cannot be accessed by following the arrows (references) from a root object are *unreachable*—and therefore subject to collection.

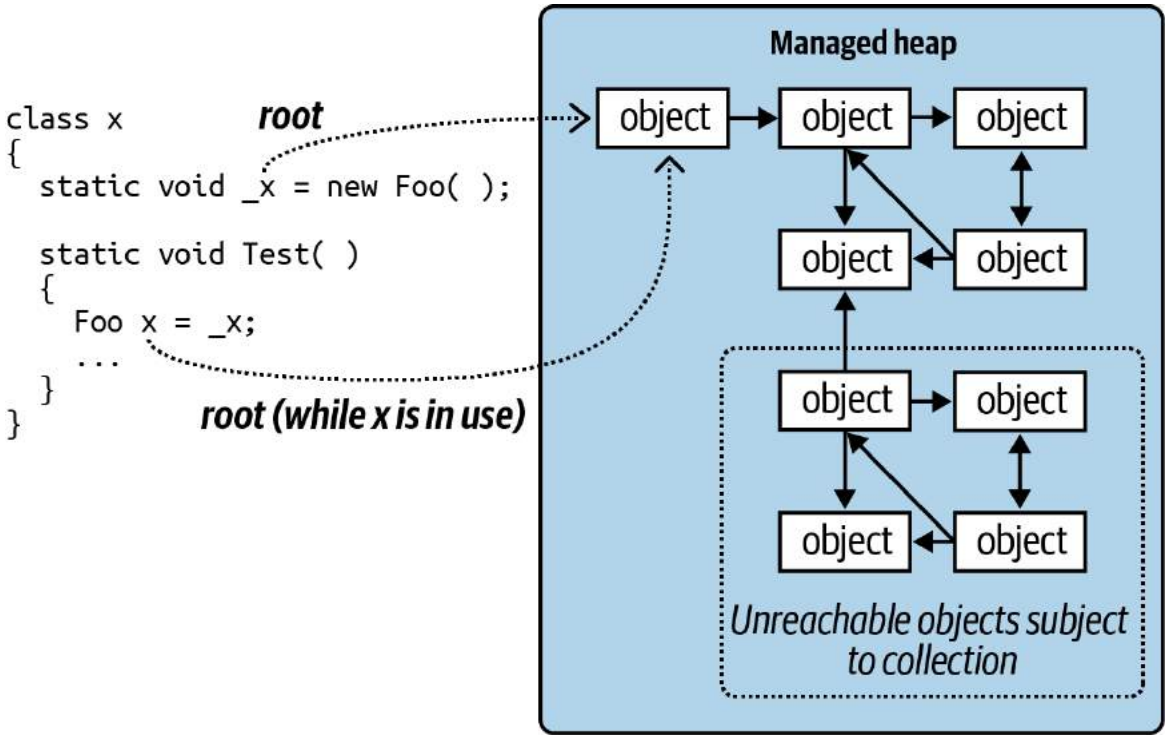

Figure 12-1. Roots

Finalizers

Prior to an object being released from memory, its *finalizer* runs, if it has one. A finalizer is declared like a constructor, but it is prefixed by the ~ symbol:

```
class Test
{
  ~Test()
  {
    // Finalizer logic...
  }
}
```

(Although similar in declaration to a constructor, finalizers cannot be declared as public or static, cannot have parameters, and cannot call the base class.)

Finalizers are possible because garbage collection works in distinct phases. First, the GC identifies the unused objects ripe for deletion. Those without finalizers are deleted immediately. Those with pending (unrun) finalizers are kept alive (for now) and are put onto a special queue.

At that point, garbage collection is complete, and your program continues executing. The *finalizer thread* then kicks in and starts running in parallel to your program, picking objects off that special queue and running their finalization methods. Prior to each object's finalizer running, it's still very much alive—that queue acts as a root object. After it's been dequeued and the finalizer executed, the object becomes orphaned and will be deleted in the next collection (for that object's *generation*).

Finalizers can be useful, but they come with some provisos:

- Finalizers slow the allocation and collection of memory (the GC needs to keep track of which finalizers have run).
- Finalizers prolong the life of the object and any *referred* objects (they must all await the next garbage truck for actual deletion).

- It's impossible to predict in what order the finalizers for a set of objects will be called.
- You have limited control over when the finalizer for an object will be called.
- If code in a finalizer blocks, other objects cannot be finalized.
- Finalizers can be circumvented altogether if an application fails to unload cleanly.

In summary, finalizers are somewhat like lawyers—although there are cases in which you really need them, in general you don't want to use them unless absolutely necessary. If you do use them, you need to be 100% sure you understand what they are doing for you.

Here are some guidelines for implementing finalizers:

- Ensure that your finalizer executes quickly.
- Never block in your finalizer (see "Blocking").
- Don't reference other finalizable objects.
- Don't throw exceptions.

> **NOTE**
>
> The CLR can call an object's finalizer even if an exception is thrown during construction. For this reason, it pays not to assume that fields are correctly initialized when writing a finalizer.

Calling Dispose from a Finalizer

A popular pattern is to have the finalizer call `Dispose`. This makes sense when cleanup is not urgent and hastening it by calling `Dispose` is more of an optimization than a necessity.

> **NOTE**
>
> Keep in mind that with this pattern you couple memory deallocation to resource deallocation—two things with potentially divergent interests (unless the resource is itself memory). You also increase the burden on the finalization thread.
>
> This pattern also serves as a backup for cases when a consumer simply forgets to call `Dispose`. However, it's then a good idea to log the failure so that you can fix the bug.

There's a standard pattern for implementing this, as follows:

```
class Test : IDisposable
{
  public void Dispose()             // NOT virtual
  {
    Dispose (true);
    GC.SuppressFinalize (this);     // Prevent finalizer from running.
  }

  protected virtual void Dispose (bool disposing)
  {
    if (disposing)
    {
      // Call Dispose() on other objects owned by this instance.
      // You can reference other finalizable objects here.
      // ...
    }

    // Release unmanaged resources owned by (just) this object.
    // ...
  }

  ~Test() => Dispose (false);
}
```

`Dispose` is overloaded to accept a `bool disposing` flag. The parameterless version is *not* declared as `virtual` and simply calls the enhanced version with `true`.

The enhanced version contains the actual disposal logic and is `protected` and `virtual`; this provides a safe point for subclasses to add their own

disposal logic. The `disposing` flag means it's being called "properly" from the `Dispose` method rather than in "last-resort mode" from the finalizer. The idea is that when called with `disposing` set to `false`, this method should not, in general, reference other objects with finalizers (because such objects might themselves have been finalized and so be in an unpredictable state). This rules out quite a lot! Here are a couple of tasks that the `Dispose` method can still perform in last-resort mode, when `disposing` is `false`:

- Releasing any *direct references* to OS resources (obtained, perhaps, via a P/Invoke call to the Win32 API)
- Deleting a temporary file created on construction

To make this robust, any code capable of throwing an exception should be wrapped in a `try`/`catch` block, and the exception, ideally, logged. Any logging should be as simple and robust as possible.

Notice that we call `GC.SuppressFinalize` in the parameterless `Dispose` method—this prevents the finalizer from running when the GC later catches up with it. Technically, this is unnecessary given that `Dispose` methods must tolerate repeated calls. However, doing so improves performance because it allows the object (and its referenced objects) to be garbage-collected in a single cycle.

Resurrection

Suppose a finalizer modifies a living object such that it refers back to the dying object. When the next garbage collection happens (for the object's generation), the CLR will see the previously dying object as no longer orphaned—and so it will evade garbage collection. This is an advanced scenario and is called *resurrection*.

To illustrate, suppose that we want to write a class that manages a temporary file. When an instance of that class is garbage-collected, we'd like the finalizer to delete the temporary file. It sounds easy:

```
public class TempFileRef
{
  public readonly string FilePath;
  public TempFileRef (string filePath) { FilePath = filePath; }

  ~TempFileRef() { File.Delete (FilePath); }
}
```

Unfortunately, this has a bug: `File.Delete` might throw an exception (due to a lack of permissions, perhaps, or the file being in use, or having already been deleted). Such an exception would take down the entire application (as well as preventing other finalizers from running). We could simply "swallow" the exception with an empty catch block, but then we'd never know that anything went wrong. Calling some elaborate error reporting API would also be undesirable because it would burden the finalizer thread, hindering garbage collection for other objects. We want to restrict finalization actions to those that are simple, reliable, and quick.

A better option is to record the failure to a static collection, as follows:

```
public class TempFileRef
{
  static internal readonly 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);   // 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.

> **NOTE**
>
> `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 some point we're going to want to dequeue items from `FailedDeletions` so that we can do something about them. This also must 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 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 words, adding it to the `FailedDeletions` queue after the third failure.

> **WARNING**
>
> 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!

How the GC Works

The standard 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* GC 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 can 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. When this process is complete, all objects that 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 prevents 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 prevents 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 OS is unable to grant further memory, an `OutOfMemoryException` is thrown.

> **NOTE**
>
> You can obtain information about the current state of the managed heap by calling `GC.GetGCMemoryInfo()`. From .NET 5, this method has been enhanced to return performance-related data.

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*. Gen0 and Gen1 are known as *ephemeral* (short-lived) generations.

The CLR keeps the Gen0 section relatively small (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 collections that include Gen2, however, take much longer and so happen infrequently. Figure 12-2 shows the effect of a full collection.

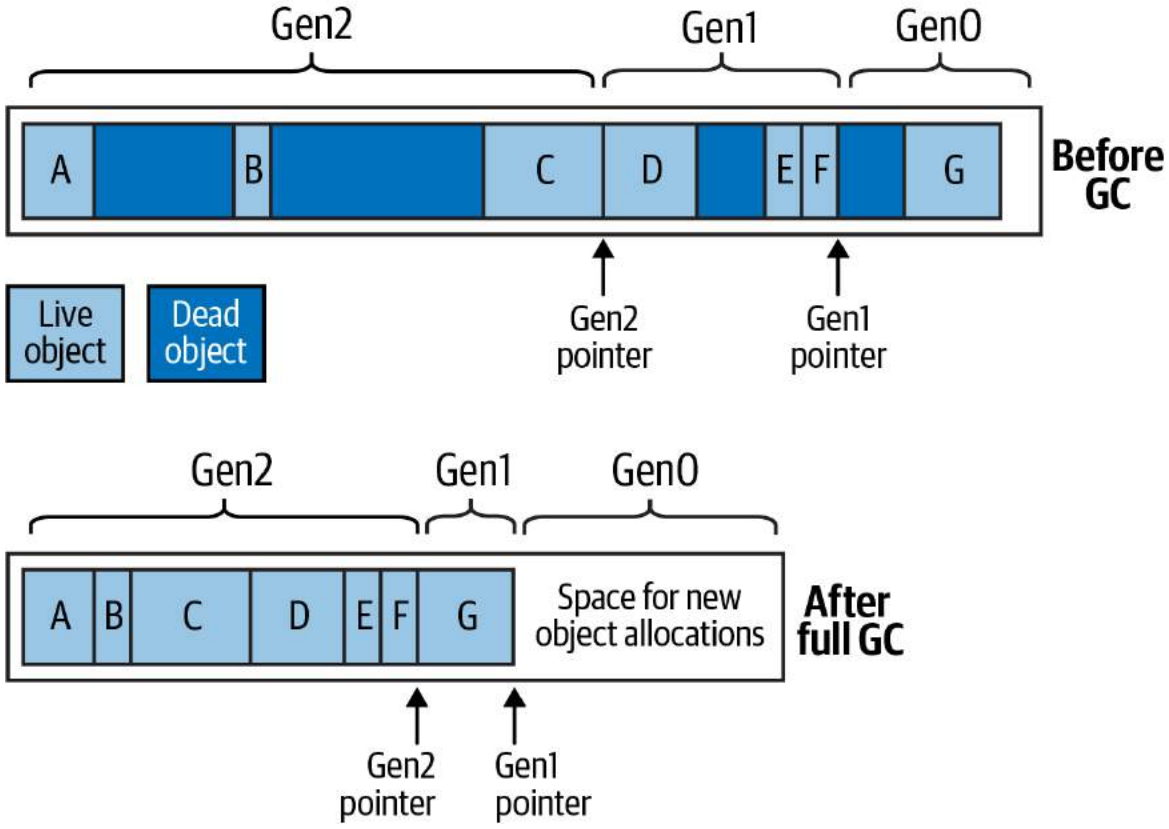

Figure 12-2. Heap generations

To give some very rough ballpark figures, a Gen0 collection might take less than one millisecond, 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 can 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 `StringBuilder`s 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");
  sb2.Append (sb1.ToString());
  return sb2.ToString();
}
```

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 prevents the cost of compacting large objects and prevents excessive Gen0 collections—without the LOH, allocating a series of 16 MB objects might trigger a Gen0 collection after every allocation.

By default, 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.[1]
- The LOH is subject to *fragmentation*. This means that the freeing of an object can create a hole in the LOH that can be difficult 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).

Should you anticipate a problem with fragmentation, you can instruct the GC to compact the LOH in the next collection, as follows:

```
GCSettings.LargeObjectHeapCompactionMode =
  GCLargeObjectHeapCompactionMode.CompactOnce;
```

Another workaround, if your program frequently allocates large arrays, is to use .NET's array pooling API (see "Array Pooling").

The LOH is also nongenerational: all objects are treated as Gen2.

Workstation versus server collection

.NET provides two garbage collection modes: *workstation* and *server*. *Workstation* is the default; you can switch to *server* by adding the following to your application's *.csproj* file:

```
<PropertyGroup>
  <ServerGarbageCollection>true</ServerGarbageCollection>
</PropertyGroup>
```

Upon building your project, this setting is written to the application's *.runtimeconfig.json* file, where's it's read by the CLR:

```
"runtimeOptions": {
  "configProperties": {
    "System.GC.Server": true
  ...
```

When server collection is enabled, the CLR allocates a separate heap and GC to each core. This speeds up collection but consumes additional memory and CPU resources (because each core requires its own thread). Should the machine be running many other processes with server collection enabled, this can lead to CPU oversubscription, which is particularly harmful on workstations because it makes the OS as a whole feel unresponsive.

Server collection is available only on multicore systems: on single-core devices (or single-core virtual machines), the setting is ignored.

Background collection

In both workstation and server modes, the CLR enables *background collection* by default. You can disable it by adding the following to your application's *.csproj* file:

```
<PropertyGroup>
  <ConcurrentGarbageCollection>false</ConcurrentGarbageCollection>
</PropertyGroup>
```

Upon building, this setting is written to the application's *.runtimeconfig.json* file:

```
"runtimeOptions": {
  "configProperties": {
    "System.GC.Concurrent": false,
 ...
```

The GC must freeze (block) your execution threads for periods during a collection. Background collection minimizes these periods of latency, making your application more responsive. This comes at the expense of consuming slightly more CPU and memory. Hence, by disabling background collection, you accomplish the following:

- Slightly reduce CPU and memory usage
- Increase the pauses (or *latency*) when a garbage collection occurs

Background collection works by allowing your application code to run in parallel with a Gen2 collection. (Gen0 and Gen1 collections are considered sufficiently fast that they don't benefit from this parallelism.)

Background collection is an improved version of what was formerly called *concurrent collection*: it 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 allows applications that continually allocate memory to be more responsive.

GC notifications

If you disable background collection, you can ask the GC to notify you just before a full (blocking) collection 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 14) 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, collection is complete, and you can again accept requests. You then repeat the whole cycle.

Forcing Garbage Collection

You can manually force a garbage collection 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.

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 (and Gen1 objects to Gen2). 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, take the additional step of calling `WaitForPendingFinalizers` and re-collecting:

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

Often this is done in a loop: the act of running finalizers can free up more objects that themselves have finalizers.

Another case for calling `GC.Collect` is when you're testing a class that has a finalizer.

Tuning Garbage Collection at Runtime

The static `GCSettings.LatencyMode` property determines how the GC balances latency with overall efficiency. Changing this from its default value of `Interactive` to either `LowLatency` or `SustainedLowLatency` instructs the CLR to favor quicker (but more frequent) collections. This is useful if your application needs to respond very quickly to real-time events. Changing the mode to `Batch` maximizes throughput at the expense of potentially poor responsiveness, which is useful for batch processing.

`SustainedLowLatency` is not supported if you disable background collection in the *.runtimeconfig.json* file.

You can also tell the CLR to temporarily suspend garbage collection by calling `GC.TryStartNoGCRegion`, and resume it with `GC.EndNoGCRegion`.

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 24), 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 instructing the CLR to *assume* that a specified quantity of unmanaged memory has been allocated; you do this by calling `GC.AddMemoryPressure`. To undo this (when the unmanaged memory is released), call `GC.RemoveMemoryPressure`.

Array Pooling

If your application frequently instantiates arrays, you can avoid most of the garbage collection overhead with *array pooling*. Array pooling was introduced in .NET Core 3 and works by "renting" an array, which you later return to a pool for reuse.

To allocate an array, call the `Rent` method on the `ArrayPool` class in the `System.Buffers` namespace, indicating the size of the array that you'd like:

```
int[] pooledArray = ArrayPool<int>.Shared.Rent (100);  // 100 bytes
```

This allocates an array of (at least) 100 bytes from the global shared array pool. The pool manager might give you an array that's larger than what you asked for (typically, it allocates in powers of 2).

When you've finished with the array, call `Return`: this releases the array to the pool, allowing the same array to be rented again:

```
ArrayPool<int>.Shared.Return (pooledArray);
```

You can optionally pass in a Boolean value instructing the pool manager to clear the array before returning it to the pool.

> **WARNING**
>
> A limitation of array pooling is that nothing prevents you from continuing to (illegally) use an array after it's been returned, so you need to code carefully to avoid this scenario. Keep in mind that you have the power to break not just your own code but other APIs that use array pooling, too, such as ASP.NET Core.

Rather than using the shared array pool, you can create a custom pool and rent from that. This prevents the risk of breaking other APIs, but increases overall memory usage (as it reduces the opportunities for reuse):

```
var myPool = ArrayPool<int>.Create();
int[] array = myPool.Rent (100);
...
```

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 must 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
{
  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` objects will become eligible for collection. Unfortunately, each client has another referee: the `_host` object whose `Click` event now references each `Client` instance. This can 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());
```

> **NOTE**
>
> In "Weak References", we describe another solution to this problem, which can be useful in environments that tend not to use disposable objects (an example is Windows Presentation Foundation [WPF]). In fact, WPF offers a class called `WeakEventManager` that uses a pattern that employs weak references.

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 that the runtime itself holds references to active timers so that it can fire their `Elapsed` events; hence:

- The runtime 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 runtime no longer references the object:

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

> **NOTE**
>
> 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 the same way with respect to what's just been discussed.

The timer in the `System.Threading` namespace, however, is special. .NET 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 timer, a finalizer can fire that will automatically stop and dispose of the timer:

```
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))
{
  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 need 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 event 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:

```
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 is collected, the `Target` property of the `WeakReference` will be null:

```
var weak = GetWeakRef();
GC.Collect();
Console.WriteLine (weak.Target);   // (nothing)

WeakReference GetWeakRef () =>
  new WeakReference (new StringBuilder ("weak"));
```

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

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

After 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.

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
{
  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 */ }
```

This strategy can 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 can 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. Assuming that is the case, you can implement a *weak delegate* class as follows:

```
public class WeakDelegate<TDelegate> where TDelegate : Delegate
{
  class MethodTarget
```

```
{
  public readonly WeakReference Reference;
  public readonly MethodInfo Method;

  public MethodTarget (Delegate d)
  {
    // d.Target will be null for static method targets:
    if (d.Target != null) Reference = new WeakReference (d.Target);
    Method = d.Method;
  }
}

List<MethodTarget> _targets = new List<MethodTarget>();

public void Combine (TDelegate target)
{
  if (target == null) return;

  foreach (Delegate d in (target as Delegate).GetInvocationList())
    _targets.Add (new MethodTarget (d));
}

public void Remove (TDelegate target)
{
  if (target == null) return;
  foreach (Delegate d in (target as Delegate).GetInvocationList())
  {
    MethodTarget mt = _targets.Find (w =>
      Equals (d.Target, w.Reference?.Target) &&
      Equals (d.Method.MethodHandle, w.Method.MethodHandle));

    if (mt != null) _targets.Remove (mt);
  }
}

public TDelegate Target
{
  get
  {
    Delegate combinedTarget = null;

    foreach (MethodTarget mt in _targets.ToArray())
    {
      WeakReference wr = mt.Reference;

      // Static target || alive instance target
      if (wr == null || wr.Target != null)
      {
```

```
            var newDelegate = Delegate.CreateDelegate (
              typeof(TDelegate), wr?.Target, mt.Method);
              combinedTarget = Delegate.Combine (combinedTarget, newDelegate);
          }
          else
            _targets.Remove (mt);
        }

        return combinedTarget as TDelegate;
      }
      set
      {
        _targets.Clear();
        Combine (value);
      }
    }
  }
```

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 of 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, removing the remaining (dead) references from the list to prevent the `_targets` list from 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 [see "Locking and Thread Safety"].) We also allow delegates without a weak reference at all; these represent delegates whose target is a static method.

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)
    => _click.Target?.Invoke (this, e);
}
```

---

1 The same thing can occur occasionally in the generational heap due to pinning (see "The fixed Statement").