Chapter 13. Diagnostics

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

Some diagnostic tools and APIs are Windows specific because they rely on features of the Windows operating system. In an effort to prevent platform-specific APIs from cluttering the .NET BCL, Microsoft has shipped them in separate NuGet packages that you can optionally reference. There are more than a dozen Windows-specific packages, which you can reference all at once with the *Microsoft.Windows.Compatibility* "master" package.

The types in this chapter are defined primarily in the `System.Diagnostics` namespace.

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). Logically, they execute before the main compilation takes place (although in practice, the compiler processes them during the lexical parsing phase). The preprocessor directives for conditional compilation are `#if`, `#else`, `#endif`, and `#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 in source code by using the `#define` directive (in which case the symbol applies to just that file), or in the *.csproj* file by using a `<DefineConstants>` element (in which case the symbol applies to 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
    Console.WriteLine ("in test mode!");     // OUTPUT: in test mode!
#endif
  }
}
```

If we deleted the first line, the program would compile with the `Console.WriteLine` statement completely eliminated from the executable, as though it were commented out.

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

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

Keep 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.

You can define symbols that apply to every file in an assembly by editing the *.csproj* file (or in Visual Studio, by going to the Build tab in the Project Properties window). The following defines two constants, `TESTMODE` and `PLAYMODE`:

```
<PropertyGroup>
  <DefineConstants>TESTMODE;PLAYMODE</DefineConstants>
</PropertyGroup>
```

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

Conditional Compilation Versus Static Variable Flags

You could instead implement the preceding example with a simple static field:

```
static internal bool TestMode = true;

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 the following:

- 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 =
  #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 runtime versions, leveraging the latest features where available.

Another advantage of conditional compilation is that debugging code can refer to 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 that 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 that you want 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
```

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 be 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 treat calls to `LogStatus` as though they were wrapped in an `#if LOGGINGMODE` directive. If the symbol is not defined, any calls to `LogStatus` are eliminated entirely in compilation—including their argument evaluation expressions. (Hence any side-effecting expressions will be bypassed.) This works even if `LogStatus` and the caller are in different assemblies.

> **NOTE**
>
> 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` are 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. (Visual Studio provides checkboxes for defining these symbols in the Build tab of Project Properties, and enables the TRACE symbol by default with new projects.)

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.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 cover this in "TraceListener").

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:

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

`Assert` simply calls `Fail` if the `bool` argument is `false`—this is called *making an assertion* and indicates a bug in the code if violated. 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 in addition to the message, which can be useful in processing the output.

An alternative to 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");
```

```
    ...
  }
```

Such “assertions” are compiled unconditionally and are less flexible in that you can’t control the outcome of a failed assertion via `TraceListeners`. And technically, they’re not assertions. An assertion is something that, if violated, indicates a bug in the current method’s code. Throwing an exception based on argument validation indicates a bug in the *caller*’s code.

TraceListener

The `Trace` class has a static `Listeners` property that returns a 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), the application is terminated.

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 (Windows only).

- `EventProviderTraceListener` writes to the Event Tracing for Windows (ETW) subsystem (cross-platform support).

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

The following example clears `Trace`'s default listener and 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"))
  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:

```
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 might 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).

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.

> **NOTE**
>
> 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 can be lost.

Debugger Integration

Sometimes, it's useful for an application to interact with a debugger if one is available. During development, the debugger is usually your IDE (e.g., Visual Studio); in deployment, the debugger is more likely to be one of the lower-level debugging tools, such as WinDbg, Cordbg, or MDbg.

Attaching and Breaking

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

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

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

Debugger Attributes

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

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

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

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

Processes and Process Threads

We described in the last section of Chapter 6 how to use `Process.Start` to launch a new process. The `Process` class also allows you to query and interact with other processes running on the same or another computer. The `Process` class is part of .NET Standard 2.0, although its features are restricted for the UWP platform.

Examining Running Processes

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

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

`Process.GetCurrentProcess` returns the current process.

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

Examining Threads in a Process

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

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

StackTrace and StackFrame

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

> **NOTE**
>
> If you just need to know the name and line number of the calling method, caller info attributes can provide an easier and faster alternative. We cover this topic in "Caller Info Attributes".

If you instantiate a `StackTrace` object with no arguments—or with a `bool` argument—you get a snapshot of the current thread's call stack. The `bool` argument, if `true`, instructs `StackTrace` to read the assembly *.pdb* (project debug) files if they are present, giving you access to filename, line number, and column offset data. Project debug files are generated when you compile with the `/debug` switch. (Visual Studio compiles with this switch unless you request otherwise via *Advanced Build Settings*.)

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

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

  Console.WriteLine ("Total frames:   " + s.FrameCount);
  Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
  Console.WriteLine ("Calling method: " + s.GetFrame(1).GetMethod().Name);
  Console.WriteLine ("Entry method:   " + s.GetFrame
                                       (s.FrameCount-1).GetMethod().Name);
```

```
    Console.WriteLine ("Call Stack:");
    foreach (StackFrame f in s.GetFrames())
      Console.WriteLine (
        "  File: "   + f.GetFileName() +
        "  Line: "   + f.GetFileLineNumber() +
        "  Col: "    + f.GetFileColumnNumber() +
        "  Offset: " + f.GetILOffset() +
        "  Method: " + f.GetMethod().Name);
  }
```

Here's the output:

```
Total frames:   4
Current method: C
Calling method: B
Entry method: Main
Call stack:
  File: C:\Test\Program.cs  Line: 15  Col: 4  Offset: 7  Method: C
  File: C:\Test\Program.cs  Line: 12  Col: 22  Offset: 6  Method: B
  File: C:\Test\Program.cs  Line: 11  Col: 22  Offset: 6  Method: A
  File: C:\Test\Program.cs  Line: 10  Col: 25  Offset: 6  Method: Main
```

NOTE

The Intermediate Language (IL) offset indicates the offset of the instruction that will execute *next*—not the instruction that's currently executing. Peculiarly, though, the line and column number (if a *.pdb* file is present) usually indicate the actual execution point.

This happens because the CLR does its best to *infer* the actual execution point when calculating the line and column from the IL offset. The compiler emits IL in such a way as to make this possible—including inserting `nop` (no-operation) instructions into the IL stream.

Compiling with optimizations enabled, however, disables the insertion of `nop` instructions, and so the stack trace might show the line and column number of the next statement to execute. Obtaining a useful stack trace is further hampered by the fact that optimization can pull other tricks, including collapsing entire methods.

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

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

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

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

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

Windows Event Logs

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

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

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

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

- Application
- System
- Security

The Application log is where most applications normally write.

Writing to the Event Log

To write to a Windows event log:

1. Choose one of the three event logs (usually *Application*).
2. Decide on a *source name* and create it if necessary (create requires administrative permissions).
3. Call `EventLog.WriteEntry` with the log name, source name, and message data.

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

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

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

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

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

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

Reading the Event Log

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

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

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

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

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

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

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

Monitoring the Event Log

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

To enable log monitoring:

1. Instantiate an `EventLog` and set its `EnableRaisingEvents` property to `true`.

2. Handle the `EntryWritten` event.

For example:

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

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

Performance Counters

> **NOTE**
>
> Performance Counters are a Windows-only feature and require the NuGet package `System.Diagnostics.PerformanceCounter`. If you're targeting Linux or macOS, see "Cross-Platform Diagnostic Tools" for alternatives.

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

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

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

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

> **NOTE**
>
> Reading performance counters or categories might require administrator privileges on the local or target computer, depending on what is accessed.

Enumerating the Available Counters

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

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

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

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

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

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

```
  var x =
    new XElement ("counters",
      from PerformanceCounterCategory cat in
           PerformanceCounterCategory.GetCategories()
      where cat.CategoryName.StartsWith (".NET")
      let instances = cat.GetInstanceNames()
      select new XElement ("category",
        new XAttribute ("name", cat.CategoryName),
        instances.Length == 0
        ?
          from c in cat.GetCounters()
          select new XElement ("counter",
            new XAttribute ("name", c.CounterName))
        :
          from i in instances
          select new XElement ("instance", new XAttribute ("name", i),
```

```
          !cat.InstanceExists (i)
          ?
            null
          :
            from c in cat.GetCounters (i)
            select new XElement ("counter",
              new XAttribute ("name", c.CounterName))
        )
    )
  );
x.Save ("counters.xml");
```

Reading Performance Counter Data

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

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

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

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

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

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

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

static void Monitor (string category, string counter, string instance,
                     EventWaitHandle stopper)
{
  if (!PerformanceCounterCategory.Exists (category))
    throw new InvalidOperationException ("Category does not exist");

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

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

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

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

```
EventWaitHandle stopper = new ManualResetEvent (false);

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

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

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

Creating Counters and Writing Performance Data

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

```
string category = "Nutshell Monitoring";

// We'll create two counters in this category:
string eatenPerMin = "Macadamias eaten so far";
string tooHard = "Macadamias deemed too hard";

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

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

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

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

The new counters then show up in the Windows performance-monitoring tool when you choose Add Counters. If you later want to define more counters in the same category, you must first delete the old category by calling `PerformanceCounterCategory.Delete`.

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

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

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

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

The Stopwatch Class

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

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

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

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

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

Cross-Platform Diagnostic Tools

In this section, we briefly describe the cross-platform diagnostic tools available to .NET:

dotnet-counters

Provides an overview of the state of a running application

dotnet-trace

For more detailed performance and event monitoring

dotnet-dump

To obtain a memory dump on demand or after a crash

These tools do not require administrative elevation and are suitable for both development and production environments.

dotnet-counters

The *dotnet-counters* tool monitors the memory and CPU usage of a .NET process and writes the data to the console (or a file).

To install the tool, run the following from a command prompt or terminal with *dotnet* in the path:

```
dotnet tool install --global dotnet-counters
```

You can then start monitoring a process, as follows:

```
dotnet-counters monitor System.Runtime --process-id <<ProcessID>>
```

`System.Runtime` means that we want to monitor all counters under the *System.Runtime* category. You can specify either a category or counter name (the `dotnet-counters list` command lists all available categories and counters).

The output is continually refreshed and looks like this:

```
Press p to pause, r to resume, q to quit.
    Status: Running

[System.Runtime]
    # of Assemblies Loaded                              63
    % Time in GC (since last GC)                         0
    Allocation Rate (Bytes / sec)                  244,864
    CPU Usage (%)                                        6
    Exceptions / sec                                     0
    GC Heap Size (MB)                                    8
    Gen 0 GC / sec                                       0
    Gen 0 Size (B)                                 265,176
    Gen 1 GC / sec                                       0
    Gen 1 Size (B)                                 451,552
    Gen 2 GC / sec                                       0
    Gen 2 Size (B)                                      24
    LOH Size (B)                                 3,200,296
    Monitor Lock Contention Count / sec                  0
    Number of Active Timers                              0
    ThreadPool Completed Work Items / sec               15
    ThreadPool Queue Length                              0
    ThreadPool Threads Count                             9
    Working Set (MB)                                    52
```

Here are all available commands:

Commands	Purpose
`list`	Displays a list of counter names along with a description of each
`ps`	Displays a list of dotnet processes eligible for monitoring
`monitor`	Displays values of selected counters (periodically refreshed)
`collect`	Saves counter information to a file

The following parameters are supported:

Options/arguments	Purpose
`--version`	Displays the version of *dotnet-counters*.
`-h, --help`	Displays help about the program.
`-p, --process-id`	ID of dotnet process to monitor. Applies to the `monitor` and `collect` commands.
`--refresh-interval`	Sets the desired refresh interval in seconds. Applies to the `monitor` and `collect` commands.
`-o, --output`	Sets the output file name. Applies to the `collect` command.
`--format`	Sets the output format. Valid are *csv* or *json*. Applies to the `collect` command.

dotnet-trace

Traces are timestamped records of events in your program, such as a method being called or a database being queried. Traces can also include performance metrics and custom events, and can contain local context such as the value of local variables. Traditionally, .NET Framework and frameworks such as ASP.NET used ETW. In .NET 5, application traces are written to ETW when running on Windows and LTTng on Linux.

To install the tool, run the following command:

```
dotnet tool install --global dotnet-trace
```

To start recording a program's events, run the following command:

```
dotnet-trace collect --process-id <<ProcessId>>
```

This runs *dotnet-trace* with the default profile, which collects CPU and .NET runtime events and writes to a file called *trace.nettrace*. You can specify other profiles with the `--profile` switch: *gc-verbose* tracks garbage collection and sampled object allocation, and *gc-collect* tracks garbage collection with a low overhead. The `-o` switch lets you specify a different output filename.

The default output is a *.netperf* file, which can be analyzed directly on a Windows machine with the PerfView tool. Alternatively, you can instruct *dotnet-trace* to create a file compatible with Speedscope, which is a free online analysis service at *https://speedscope.app*. To create a Speedscope (*.speedscope.json*) file, use the option `--format speedscope`.

NOTE

You can download the latest version of PerfView from *https://github.com/microsoft/perfview*. The version that ships with Windows 10 might not support *.netperf* files.

The following commands are supported:

Commands	Purpose
`collect`	Starts recording counter information to a file.
`ps`	Displays a list of dotnet processes eligible for monitoring.
`list-profiles`	Lists prebuilt tracing profiles with a description of providers and filters in each.
`convert <file>`	Converts from the *nettrace* (*.netperf*) format to an alternative format. Currently, *speedscope* is the only target option.

Custom trace events

Your app can emit custom events by defining a custom `EventSource`:

```
[EventSource (Name = "MyTestSource")]
public sealed class MyEventSource : EventSource
{
  public static MyEventSource Instance = new MyEventSource ();

  MyEventSource() : base (EventSourceSettings.EtwSelfDescribingEventFormat)
  {
  }

  public void Log (string message, int someNumber)
  {
    WriteEvent (1, message, someNumber);
  }
}
```

The `WriteEvent` method is overloaded to accept various combinations of simple types (primarily strings and integers). You can then call it as follows:

```
MyEventSource.Instance.Log ("Something", 123);
```

When calling *dotnet-trace*, you must specify the name(s) of any custom event sources that want to record:

```
dotnet-trace collect --process-id <<ProcessId>> --providers MyTestSource
```

dotnet-dump

A *dump*, sometimes called a *core dump*, is a snapshot of the state of a process's virtual memory. You can dump a running process on demand, or configure the OS to generate a dump when an application crashes.

On Ubuntu Linux, the following command enables a core dump upon application crash (the necessary steps can vary between different flavors of Linux):

```
ulimit -c unlimited
```

On Windows, use *regedit.exe* to create or edit the following key in the local machine hive:

```
SOFTWARE\Microsoft\Windows\Windows Error Reporting\LocalDumps
```

Under that, add a key with the same name as your executable (e.g., *foo.exe*), and under that key, add the following keys:

- `DumpFolder` (REG_EXPAND_SZ), with a value indicating the path to which you want dump files written
- `DumpType` (REG_DWORD), with a value of 2 to request a full dump
- (Optionally) `DumpCount` (REG_DWORD), indicating the maximum number of dump files before the oldest is removed

To install the tool, run the following command:

```
dotnet tool install --global dotnet-dump
```

After you’ve installed it, you can initiate a dump on demand (without ending the process), as follows:

```
dotnet-dump collect --process-id <<YourProcessId>>
```

The following command starts an interactive shell for analyzing a dump file:

```
dotnet-dump analyze <<dumpfile>>
```

If an exception took down the application, you can use the *printexceptions* command (*pe* for short) to display details of that exception. The dotnet-dump shell supports numerous additional commands, which you can list with the *help* command.