CHAPTER 25

Interoperating with WinRT Components

In this chapter:

Windows 8 comes with a new class library allowing applications to access operating system functionality. The formal name for this class library is the *Windows Runtime (WinRT)* and its components are accessible using the WinRT type system. WinRT has many of the same goals that the Common Language Runtime (CLR) had when it was first introduced, such as simplifying application development and allowing code implemented in different programming languages to easily interoperate with one another. Specifically, Microsoft supports consuming WinRT components from native C/C++, JavaScript (when using Microsoft's "Chakra" JavaScript virtual machine), as well as C# and Visual Basic via the CLR.

Figure 25-1 shows the kinds of features exposed by Windows's WinRT components and the various languages that Microsoft supports to access them. For applications implemented with native C/C++, the developer must compile his or her code for each CPU architecture (x86, x64, and ARM). But Microsoft .NET Framework developers need to compile their code just once into Intermediate Language (IL) and then the CLR compiles that into the native code specific to the host CPU. JavaScript developers actually ship the source code to their application and the "Chakra" virtual machine parses the source code and compiles it into native code specific to the host machine's CPU. Other companies can produce languages and environments that support interoperating with WinRT components too.

Windows Store Apps and desktop applications can leverage operating system functionality by using these WinRT components. Today, the number of WinRT components that ship as part of Windows is relatively tiny when compared to the size of the .NET Framework's class library. However, this is by design, because the components are focused on exposing what an operating system does best: abstracting hardware and cross-application facilities to application developers. So, most of the WinRT components expose features, such as storage, networking, graphics, media, security, threading, and so on. Other core language services (like string manipulation) and more complex frameworks (like language integrated query) are not offered by the operating system and are instead provided by the language being used to access the operating system's WinRT components.

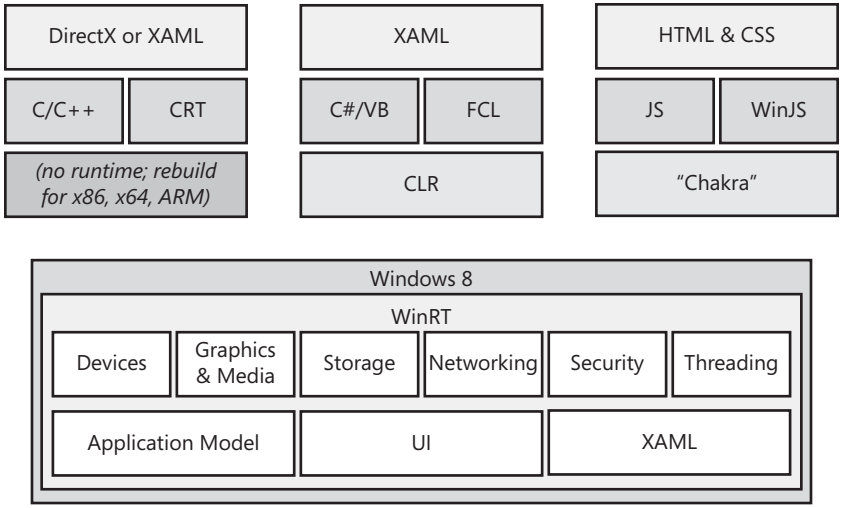

FIGURE 25-1 The kinds of features exposed by Windows's WinRT components and the various languages that Microsoft supports to access them.

WinRT components are internally implemented as Component Object Model (COM) components, which is a technology that Microsoft introduced in 1993. At the time, COM was considered a complicated model with a lot of arcane rules and a very tedious programming model. However, there are a lot of good ideas in COM, and over the years, Microsoft has tweaked it in an effort to greatly simplify it. For WinRT components, Microsoft made a very significant tweak: instead of using type libraries to describe a COM component's API, they now use metadata. That's right, WinRT components describe their API using the same .NET metadata format (ECMA-335) as was standardized by the ECMA committee. This is the same metadata format I've been discussing throughout this whole book.

Metadata is much richer than type libraries and the CLR already has a complete understanding of metadata. In addition, the CLR has supported interoperating with COM components via Runtime Callable Wrappers (RCWs) and COM Callable Wrappers (CCWs) because its inception. For the most part, this allows languages (like C#) running on top of the CLR to seamlessly interoperate with WinRT types and components.

In C#, when you have a reference to a WinRT object, you really have a reference to an RCW that internally refers to the WinRT object. Similarly, if you pass a CLR object to a WinRT API, you are really passing a reference to a CCW to the WinRT API and the CCW holds a reference to your CLR object.

WinRT components have their metadata embedded in files with a .winmd file extension. The WinRT components that ship with Windows have their metadata in the various Windows.*.winmd files, which can be found in the %WinDir%\System32\WinMetadata directory. When building an app, you would reference the one Windows.winmd file that the Windows SDK installs here.

%ProgramFiles(x86)%\Windows Kits\8.0\References\CommonConfiguration\Neutral\Windows.winmd

A major design goal of the Windows Runtime type system is to enable developers to be successful in writing apps by using the technologies, tools, practices, and conventions that are already familiar and well-known to them. In order to achieve this, some WinRT features are projected to the respective development technologies. For .NET Framework developers, there are two kinds of projections:

- **CLR projections** CLR projections are mappings performed implicitly by the CLR, usually related to reinterpreting metadata. The next section focuses on the WinRT Component Type System Rules and how the CLR projects these rules to the .NET Framework developer.
- **Framework projections** Framework projections are mappings performed explicitly in your code by leveraging new APIs introduced into the Framework Class Library. Framework projections are required when the impedance mismatch between the WinRT type system and the CLR's type system is too great for the CLR to do it implicitly. Framework projections are discussed later in this chapter.

CLR Projections and WinRT Component Type System Rules

WinRT components conform to a type system similar to how the CLR enforces a type system. When the CLR sees a WinRT type, it usually allows that type to be used via the CLRs normal COM interop technologies. But, in some cases, the CLR hides the WinRT type (by dynamically setting it to private) and then the CLR exposes the type via a different type. Internally, the CLR is looking for certain types (via metadata) and then mapping these types to types in the Framework Class Library. For the complete list of WinRT types that the CLR implicitly projects to Framework Class Library types, see *http://msdn.microsoft.com/en-us/library/windows/apps/hh995050.aspx*.

WinRT Type System Core Concepts

The WinRT type system is not as feature rich as the CLR's type system. This bulleted list describes the WinRT type system's core concepts and how the CLR projects them:

- **File names and namespaces** The name of the .winmd file itself must match the name of the namespace containing the WinRT components. For example, a file named Wintellect.WindowsStore.winmd must have WinRT components defined in a `Wintellect.WindowsStore` namespace or in a sub-namespace of `Wintellect.WindowsStore`. Because the Windows file system is case insensitive, namespaces that differ by case only are not allowed. Also, a WinRT component cannot have the same name as a namespace.
- **Common base type** WinRT components do not share a common base class. When the CLR projects a WinRT type, it appears as if the WinRT type is derived from `System.Object` and therefore all WinRT types inherit public methods like `ToString`, `GetHashCode`, `Equals`, and `GetType`. So, when using a WinRT object via C#, the object will look like it is derived from `System.Object`, and you can pass WinRT objects throughout your code. You can also call the "inherited" methods such as `ToString`.

- **Core data types** The WinRT type system supports the core data types such as Boolean, unsigned byte, 16-bit, 32-bit, and 64-bit signed and unsigned integer numbers, single-precision and double-precision floating-point numbers, 16-bit character, strings, and void.[1] Like in the CLR, all other data types are composed from these core data types.

- **Classes** WinRT is an object-oriented type system, meaning that WinRT components support data abstraction, inheritance, and polymorphism.[2] However, some languages (like JavaScript) do not support type inheritance and in order to cater to these languages, almost no WinRT components take advantage of inheritance. This means they also do not take advantage of polymorphism. In fact, only WinRT components consumable from non-JavaScript languages leverage inheritance and polymorphism. For the WinRT components that ship with Windows, only the XAML components (for building user interfaces) take advantage of inheritance and polymorphism. Applications written in JavaScript use HTML and CSS to produce their user interface instead.

- **Structures** WinRT supports structures (value types), and instances of these are marshaled by value across the COM interoperability boundary. Unlike CLR value types, WinRT structures can only have public fields of the core data types or of another WinRT structure.[3] In addition, WinRT structures cannot have any constructors or helper methods. For convenience, the CLR projects some operating system WinRT structures as some native CLR types, which do offer constructors and helper methods. These projected types feel more natural to the CLR developer. Examples include the `Point`, `Rect`, `Size`, and `TimeSpan` structures all defined in the `Windows.Foundation` namespace.

- **Nullable Structures** WinRT APIs can expose nullable structures (value types). The CLR projects the WinRT's `Windows.Foundation.IReference<T>` interface as the CLR's `System.Nullable<T>` type.

- **Enumerations** An enumeration value is simply passed as a signed or unsigned 32-bit integer. If you define an enumeration type in C#, the underlying type must be either `int` or `uint`. Also, signed 32-bit integer enums are considered to be discreet values, whereas unsigned 32-bit enums are considered to be flags capable of being OR'd together.

- **Interfaces** A WinRT interface's members must specify only WinRT-compatible types for parameters and return types.

- **Methods** WinRT has limited support for method overloading. Specifically, because JavaScript has dynamic typing, it can't distinguish between methods that differ only by the types of their parameters. For example, JavaScript will happily pass a number to a method expecting a string. However, JavaScript can distinguish between a method that takes one parameter and a method that takes two parameters. In addition, WinRT does not support operator overload methods and default argument values. Furthermore, arguments can only be marshaled in or

[1] Signed byte is not supported by WinRT.

[2] Data abstraction is actually enforced, because WinRT classes are not allowed to have public fields.

[3] Enumerations are also OK, because they are really just 32-bit integers.

out; never in and out. This means you can't apply ref to a method argument but out is OK. For more information about this, see the "Arrays" bullet point in the next list.

- **Properties** WinRT properties must specify only WinRT-compatible types for their data type. WinRT does not support parameterful properties or write-only properties.

- **Delegates** WinRT delegate types must specify only WinRT components for parameter types and return types. When passing a delegate to a WinRT component, the delegate object is wrapped with a CCW and will not get garbage collected until the CCW is released by the WinRT component consuming it. WinRT delegates do not have `BeginInvoke` and `EndInvoke` methods.

- **Events** WinRT components can expose events by using a WinRT delegate type. Because most WinRT components are sealed (no inheritance), WinRT defines a `TypedEventHandler` delegate where the sender parameter is a generic type (as opposed to `System.Object`).

```
public delegate void TypedEventHandler<TSender, TResult>(TSender sender, TResult args);
```

There is also a `Windows.Foundation.EventHandler<T>` WinRT delegate type that the CLR projects as the .NET Framework's familiar `System.EventHandler<T>` delegate type.

- **Exceptions** Under the covers, WinRT components, like COM components, indicate their status via HRESULT values (a 32-bit integer with special semantics). The CLR projects WinRT values of type `Windows.Foundation.HResult` as exception objects. When a WinRT API returns a well-known failure HRESULT value, the CLR throws an instance of a corresponding `Exception`-derived class. For instance, the HRESULT 0x8007000e (E_OUTOFMEMORY) is mapped to a `System.OutOfMemoryException`. Other HRESULT values cause the CLR to throw a `System.Exception` object whose `HResult` property contains the HRESULT value. A WinRT component implemented in C# can just throw an exception of a desired type and the CLR will convert it to an appropriate HRESULT value. To have complete control over the HRESULT value, construct an exception object, assign a specific HRESULT value in the object's `HResult` property, and then throw the object.

- **Strings** Of course, you can pass immutable strings between the WinRT and CLR type systems. However, the WinRT type system doesn't allow a string to have a value of `null`. If you pass `null` to a string parameter of a WinRT API, the CLR detects this and throws an `ArgumentNullException`; instead, use `String.Empty` to pass an empty string into a WinRT API. Strings are passed by reference to a WinRT API; they are pinned on the way in and unpinned upon return. `Strings` are always copied when returned from a WinRT API back to the CLR. When passing a CLR string array (`String[]`) to or from a WinRT API, a copy of the array is made with all its string elements and the copy is passed or returned to the other side.

- **Dates and Times** The WinRT `Windows.Foundation.DateTime` structure represents a UTC date/time. The CLR projects the WinRT `DateTime` structure as the .NET Framework's `System.DateTimeOffset` structure, because `DateTimeOffset` is preferred over the .NET Framework's `System.DateTime` structure. The CLR converts the UTC date/time being

returned from a WinRT to local time in the resulting `DateTimeOffset` instance. The CLR passes a `DateTimeOffet` to a WinRT API as a UTC time.

- **URIs** The CLR projects the WinRT `Windows.Foundation.Uri` type as the .NET Framework's `System.Uri` type. When passing a .NET Framework Uri to a WinRT API, the CLR throws an `ArgumentException` if the URI is a relative URI; WinRT supports absolute URIs only. URIs are always copied across the interop boundary.

- **IClosable/IDisposable** The CLR projects the WinRT `Windows.Foundation.IClosable` interface (which has only a `Close` method) as the .NET Framework's `System.IDisposable` interface (with its `Dispose` method). One thing to really take note of here is that all WinRT APIs that perform I/O operations are implemented asynchronously. Because `IClosable` interface's method is called `Close` and is not called `CloseAsync`, the `Close` method must not perform any I/O operations. This is semantically different from how `Dispose` usually works in the .NET Framework. For .NET Framework-implemented types, calling `Dispose` can do I/O and, in fact, it frequently causes buffered data to be written before actually closing a device. When C# code calls `Dispose` on a WinRT type however, I/O (like writing buffered data) will not be performed and a loss of data is possible. You must be aware of this and, for WinRT components that wrap output streams, you will have to explicitly call methods to prevent data loss. For example, when using a `DataWriter`, you should always call its `StoreAsync` method.

- **Arrays** WinRT APIs support single-dimension, zero-based arrays. WinRT can marshal an array's elements in or out of a method; never in and out. Because of this, you cannot pass an array into a WinRT API, have the API modify the array's elements and then access the modified elements after the API returns.[4] I have just described the contract that should be adhered to. However, this contract is not actively enforced, so it is possible that some projections might marshal array contents both in and out. This usually happens naturally due to improving performance. For example, if the array contains structures, the CLR will simply pin the array, pass it to the WinRT API, and then unpin it upon return. In effect, the array's contents are passed in, the WinRT API can modify the contents and, in effect, the modified contents are returned. However, in this example, the WinRT API is violating the contract and this behavior is not guaranteed to work. And, in fact, it will not work if the API is invoked on a WinRT component that is running out-of-process.

- **Collections** When passing a collection to a WinRT API, the CLR wraps the collection object with a CCW and passes a reference to the CCW to the WinRT API. When WinRT code invokes a member on the CCW, the calling thread crosses the interop boundary, thereby incurring a performance hit. Unlike arrays, this means that passing a collection to a WinRT API allows the API to manipulate the collection in place, and copies of the collection's elements are not being created. Table 25-1 shows the WinRT collection interfaces and how the CLR projects them to .NET application code.

[4] This means you can't have an API like `System.Array`'s `Sort` method. Interestingly, all the languages (C, C++, C#, Visual Basic, and JavaScript) support passing array elements in and out, but the WinRT type system does not allow this.

TABLE 25-1 WinRT Collection Interfaces and Projected CLR Collection Types

WinRT Collection Type (`Windows.Foundation.Collections` namespace)	Projected CLR Collection Type (`System.Collections.Generic` namespace)
`IIterable<T>`	`IEnumerable<T>`
`IVector<T>`	`IList<T>`
`IVectorView<T>`	`IReadOnlyList<T>`
`IMap<K, V>`	`IDictionary<TKey, TValue>`
`IMapView<K, V>`	`IReadOnlyDictionary<TKey, TValue>`
`IKeyValuePair<K, V>`	`KeyValuePair<TKey, TValue>`

As you can see from the previous list, the CLR team has done a lot of work to make interoperating between the WinRT type system and the CLR's type system as seamless as possible so that it is easy for managed code developers to leverage WinRT components in their code.[5]

Framework Projections

When the CLR can't implicitly project a WinRT type to the .NET Framework developer, the developer must resort to explicitly using framework projections. There are three main technologies where framework projections are required: asynchronous programming, interoperating between WinRT streams and .NET Framework streams, and when passing blocks of data between the CLR and WinRT APIs. These three framework projections are discussed in the following three sections of this chapter. Because many applications require the use of these technologies, it is important that you understand them well and use them effectively.

Calling Asynchronous WinRT APIs from .NET Code

When a thread performs an I/O operation synchronously, the thread can block for an indefinite amount of time. When a GUI thread waits for a synchronous I/O operation to complete, the application's user interface stops responding to user input, such as touch, mouse, and stylus events, causing the user to get frustrated with the application. To prevent non-responsive applications, WinRT components that perform I/O operations expose the functionality via asynchronous APIs. In fact, WinRT components that perform compute operations also expose this functionality via asynchronous APIs if the CPU operation could take greater than 50 milliseconds. For more information about building responsive applications, see Part V, "Threading" of this book.

[5] To learn even more, go to *http://msdn.microsoft.com/en-us/library/windows/apps/hh995050.aspx* and then download the CLRandtheWindowsRuntime.docx document.

Because so many WinRT APIs are asynchronous, being productive with them requires that you understand how to interoperate with them from C#. To understand it, examine the following code.

```
public void WinRTAsyncIntro() {
   IAsyncOperation<StorageFile> asyncOp = KnownFolders.MusicLibrary.GetFileAsync("Song.mp3");
   asyncOp.Completed = OpCompleted;
   // Optional: call asyncOp.Cancel() sometime later
}

// NOTE: Callback method executes via GUI or thread pool thread:
private void OpCompleted(IAsyncOperation<StorageFile> asyncOp, AsyncStatus status) {
   switch (status) {
      case AsyncStatus.Completed:    // Process result
         StorageFile file = asyncOp.GetResults(); /* Completed... */ break;

      case AsyncStatus.Canceled:     // Process cancellation
         /* Canceled... */ break;

      case AsyncStatus.Error:        // Process exception
         Exception exception = asyncOp.ErrorCode; /* Error... */ break;
   }
   asyncOp.Close();
}
```

The `WinRTAsyncIntro` method invokes the WinRT `GetFileAsync` method to find a file in the user's music library. All WinRT APIs that perform asynchronous operations are named with the `Async` suffix and they all return an object whose type implements a WinRT `IAsyncXxx` interface; in this example, an `IAsyncOperation<TResult>` interface where `TResult` is the WinRT `StorageFile` type. This object, whose reference I put in an `asyncOp` variable, represents the pending asynchronous operation. Your code must somehow receive notification when the pending operation completes. To do this, you must implement a callback method (`OpCompleted` in my example), create a delegate to it, and assign the delegate to the `asyncOp`'s `Completed` property. Now, when the operation completes, the callback method is invoked via some thread (not necessarily the GUI thread). If the operation completed before assigning the delegate to the `OnCompleted` property, then the system will invoke the callback as soon as possible. In other words, there is a race condition here, but the object implementing the `IAsyncXxx` interface resolves the race for you, ensuring that your code works correctly.

As noted at the end of the `WinRTAsyncIntro` method, you can optionally call a `Cancel` method offered by all `IAsyncXxx` interfaces if you want to cancel the pending operation. All asynchronous operations complete for one of three possible reasons: the operation runs to completion successfully, the operation is explicitly canceled, or the operation results in a failure. When the operation completes due to any of these reasons, the system invokes the callback method, passes it a reference to the same object that the original `XxxAsync` method returned, and an `AsyncStatus`. In my `OnCompleted` method, I examine the status parameter and either process the result due to the successful completion, handle the explicit cancellation, or handle the failure.[6] Also, note that after

[6] The `IAsyncInfo` interface offers a `Status` property that contains the same value that is passed into the callback method's status parameter. Because the parameter is passed by value, your application's performance is better if you access the parameter as opposed to querying `IAsyncInfo`'s `Status` property because querying the property invokes a WinRT API via an RCW.

processing the operation's completion, the IAsyncXxx interface object should be cleaned up by calling its Close method.

Figure 25-2 shows the various WinRT IAsyncXxx interfaces. The four main interfaces all derive from the IAsyncInfo interface. The two IAsyncAction interfaces give you a way to know when the operation has completed, but their operations complete with no return value (their GetResults methods have a void return type). The two IAsyncOperation interfaces also give you a way to know when the operation has completed and allow you to get their return value (their GetResults methods have a generic TResult return type).

The two IAsyncXxxWithProgress interfaces allow your code to receive periodic progress updates as the asynchronous operation is progressing through its work. Most asynchronous operations do not offer progress updates but some (like background downloading and uploading) do. To receive periodic progress updates, you would define another callback method in your code, create a delegate that refers to it, and assign the delegate to the IAsyncXxxWithProgress object's Progress property. When your callback method is invoked, it is passed an argument whose type matches the generic TProgress type.

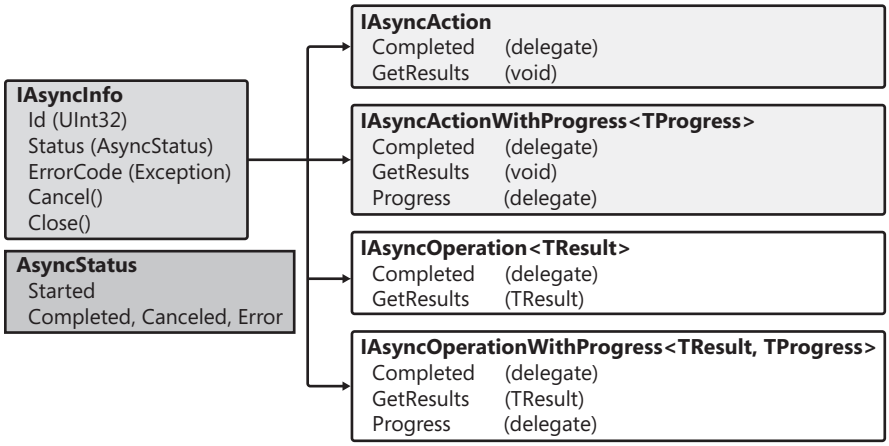

FIGURE 25-2 WinRT's interfaces related to performing asynchronous I/O and compute operations.

In the .NET Framework, we use the types in the System.Threading.Tasks namespace to simplify performing asynchronous operations. I explain these types and how to use them to perform compute operations in Chapter 27, "Compute-Bound Asynchronous Operations," and how to use them to perform I/O operations in Chapter 28, "I/O-Bound Asynchronous Operations." In addition, C# offers the async and await keywords, which allow you to perform asynchronous operations by using a sequential programming model, thereby simplifying your code substantially.

The following code is a rewrite of the WinRTAsyncIntro method previously mentioned. However, this version is leveraging some extension methods supplied with the .NET Framework, which turn the WinRT asynchronous programming model into the more convenient C# programming model.

```
using System;       // Required for extension methods in WindowsRuntimeSystemExtensions
.
.
.
public async void WinRTAsyncIntro() {
   try {
      StorageFile file = await KnownFolders.MusicLibrary.GetFileAsync("Song.mp3");
      /* Completed... */
   }
   catch (OperationCanceledException) { /* Canceled... */ }
   catch (SomeOtherException ex) { /* Error... */ }
}
```

What's happening here is that the use of C#'s await operator causes the compiler to look for a GetAwaiter method on the IAsyncOperation<StorageFile> interface returned from the GetFileAsync method. This interface doesn't provide a GetAwaiter method, and so, the compiler looks for an extension method. Fortunately, the .NET Framework team has provided in System.Runtime.WindowsRuntime.dll a bunch of extension methods callable when you have one of WinRT's IAsyncXxx interfaces.

```
namespace System {
   public static class WindowsRuntimeSystemExtensions {
      public static TaskAwaiter GetAwaiter(this IAsyncAction source);
      public static TaskAwaiter GetAwaiter<TProgress>(this IAsyncActionWithProgress<TProgress>
source);
      public static TaskAwaiter<TResult> GetAwaiter<TResult>(this IAsyncOperation<TResult>
source);
      public static TaskAwaiter<TResult> GetAwaiter<TResult, TProgress>(
         this IAsyncOperationWithProgress<TResult, TProgress> source);
   }
}
```

Internally, all these methods construct a TaskCompletionSource and tell the IAsyncXxx object to invoke a callback that sets the TaskCompletionSource's final state when the asynchronous operation completes. The TaskAwaiter object returned from these extension methods is ultimately what C# awaits. When the asynchronous operation completes, the TaskAwaiter object ensures that the code continues executing via the SynchronizationContext (discussed in Chapter 28) that is associated with the original thread. Then, the thread executes the C# compiler generated code, which queries the TaskCompletionSource's Task's Result property, which returns the result (a StorageFile in my example), throws an OperationCanceledException in case of cancellation, or throws some other exception if a failure occurred. For an example of how these methods work internally, see the code at the end of this section.

What I have just shown is the common scenario of calling an asynchronous WinRT API and discovering its outcome. But, in the previous code, I showed how to find out if cancellation occurred, but I didn't show how to actually cancel the operation. In addition, I didn't show how to deal with progress updates. To properly handle cancellation and progress updates, instead of having the compiler automatically call one of the `GetAwaiter` extension methods shown earlier, you would explicitly call one of the `AsTask` extension methods that the `WindowsRuntimeSystemExtensions` class also defines.

```
namespace System {
   public static class WindowsRuntimeSystemExtensions {
      public static Task AsTask<TProgress>(this IAsyncActionWithProgress<TProgress> source,
         CancellationToken cancellationToken, IProgress<TProgress> progress);

      public static Task<TResult> AsTask<TResult, TProgress>(
         this IAsyncOperationWithProgress<TResult, TProgress> source,
         CancellationToken cancellationToken, IProgress<TProgress> progress);

      // Simpler overloads not shown here
   }
}
```

So now, let me show you the complete picture. Here is how to call an asynchronous WinRT API and fully leverage cancellation and progress for those times when you need these enhancements.

```
using System;               // For WindowsRuntimeSystemExtensions's AsTask
using System.Threading;     // For CancellationTokenSource

internal sealed class MyClass {
   private CancellationTokenSource m_cts = new CancellationTokenSource();

   // NOTE: If invoked by GUI thread, all code executes via GUI thread:
   private async void MappingWinRTAsyncToDotNet(WinRTType someWinRTObj) {
      try {
         // Assume XxxAsync returns IAsyncOperationWithProgress<IBuffer, UInt32>
         IBuffer result = await someWinRTObj.XxxAsync(...)
            .AsTask(m_cts.Token, new Progress<UInt32>(ProgressReport));
         /* Completed... */
      }
      catch (OperationCanceledException) { /* Canceled... */ }
      catch (SomeOtherException)         { /* Error...    */ }
   }

   private void ProgressReport(UInt32 progress) { /* Update progress... */ }

   public void Cancel() { m_cts.Cancel(); } // Called sometime later
}
```

I know that some readers would like to understand how these AsTask methods internally convert a WinRT IAsyncXxx into a .NET Framework Task that can ultimately be awaited. The following code shows how the most complicated AsTask method is effectively implemented internally. The simpler overloads are, of course, simpler than this.

```
public static Task<TResult> AsTask<TResult, TProgress>(
   this IAsyncOperationWithProgress<TResult, TProgress> asyncOp,
   CancellationToken ct = default(CancellationToken),
   IProgress<TProgress> progress = null) {

   // When CancellationTokenSource is canceled, cancel the async operation
   ct.Register(() => asyncOp.Cancel());

   // When the async operation reports progress, report it to the progress callback
   asyncOp.Progress = (asyncInfo, p) => progress.Report(p);

   // This TaskCompletionSource monitors the async operation's completion
   var tcs = new TaskCompletionSource<TResult>();

   // When the async operation completes, notify the TaskCompletionSource
   // Code awaiting the TaskCompletionSource regains control when this happens
   asyncOp.Completed = (asyncOp2, asyncStatus) => {
      switch (asyncStatus) {
         case AsyncStatus.Completed: tcs.SetResult(asyncOp2.GetResults()); break;
         case AsyncStatus.Canceled: tcs.SetCanceled(); break;
         case AsyncStatus.Error: tcs.SetException(asyncOp2.ErrorCode); break;
      }
   };

   // When calling code awaits this returned Task, it calls GetAwaiter, which
   // wraps a SynchronizationContext around the Task ensuring that completion
   // occurs on the SynchronizationContext object's context
   return tcs.Task;
}
```

Interoperating Between WinRT Streams and .NET Streams

There are many .NET Framework classes that operate on System.IO.Stream-derived types, such as serialization and LINQ to XML. To use a WinRT object that implements WinRT's IStorageFile or IStorageFolder interfaces with a .NET Framework class that requires a Stream-derived type, we leverage extension methods defined in the System.IO.WindowsRuntimeStorageExtensions class.

```
namespace System.IO {   // Defined in System.Runtime.WindowsRuntime.dll
   public static class WindowsRuntimeStorageExtensions {
      public static Task<Stream> OpenStreamForReadAsync(this IStorageFile file);
      public static Task<Stream> OpenStreamForWriteAsync(this IStorageFile file);

      public static Task<Stream> OpenStreamForReadAsync(this IStorageFolder rootDirectory,
         String relativePath);
      public static Task<Stream> OpenStreamForWriteAsync(this IStorageFolder rootDirectory,
         String relativePath, CreationCollisionOption creationCollisionOption);
   }
}
```

Here is an example that uses one of the extension methods to open a WinRT `StorageFile` and read its contents into a .NET Framework `XElement` object.

```
async Task<XElement> FromStorageFileToXElement(StorageFile file) {
   using (Stream stream = await file.OpenStreamForReadAsync()) {
      return XElement.Load(stream);
   }
}
```

Finally, the `System.IO.WindowsRuntimeStreamExtensions` class offers extension methods that "cast" WinRT stream interfaces (such as `IRandomAccessStream`, `IInputStream` or `IOutputStream`) to the .NET Framework's `Stream` type and vice versa.

```
namespace System.IO {   // Defined in System.Runtime.WindowsRuntime.dll
   public static class WindowsRuntimeStreamExtensions {
      public static Stream AsStream(this IRandomAccessStream winRTStream);
      public static Stream AsStream(this IRandomAccessStream winRTStream, Int32 bufferSize);

      public static Stream AsStreamForRead(this IInputStream winRTStream);
      public static Stream AsStreamForRead(this IInputStream winRTStream, Int32 bufferSize);

      public static Stream AsStreamForWrite(this IOutputStream winRTStream);
      public static Stream AsStreamForWrite(this IOutputStream winRTStream, Int32 bufferSize);

      public static IInputStream  AsInputStream (this Stream clrStream);
      public static IOutputStream AsOutputStream(this Stream clrStream);
   }
}
```

Here is an example that uses one of the extension methods to "cast" a WinRT `IInputStream` to a .NET Framework Stream object.

```
XElement FromWinRTStreamToXElement(IInputStream winRTStream) {
   Stream netStream = winRTStream.AsStreamForRead();
   return XElement.Load(netStream);
}
```

Note that the "casting" extension methods provided by the .NET Framework do a little more than just casting under the covers. Specifically, the methods that adapt a WinRT stream to a .NET Framework stream implicitly create a buffer for the WinRT stream in the managed heap. As a result, most operations write to this buffer and do not need to cross the interop boundary, thereby improving performance. This is especially significant in scenarios that involve many small I/O operations, such as parsing an XML document.

One of the benefits of using the .NET Framework's stream projections is that if you use an `AsStreamXxx` method more than once on the same WinRT stream instance, you do not need to worry that different, disconnected buffers will be created, and data written to one buffer will not be visible in the other. The .NET Framework APIs ensure that every stream object has a unique adapter instance and all users share the same buffer.

Although in most cases the default buffering offers a good compromise between performance and memory usage, there are some cases where you may want to tweak the size of your buffer to be different from the 16-KB default. The `AsStreamXxx` methods offer overloads for that. For instance, if you know that you will be working with a very large file for an extended period of time, and that not many other buffered streams will be used at the same time, you can gain some additional performance by requesting a very large buffer for your stream. Conversely, in some network scenarios with low-latency requirements, you may want to ensure that no more bytes are read from the network than were explicitly requested by your application. In such cases, you can disable buffering altogether. If you specify a buffer size of zero bytes to the `AsStreamXxx` methods, no buffer object will be created.

Passing Blocks of Data Between the CLR and WinRT

When possible, you should use the stream framework projections as discussed in the previous section, because they have pretty good performance characteristics. However, there may be times when you need to pass raw blocks of data between the CLR and WinRT components. For example, WinRT's file and socket stream components require you to write and read raw blocks of data. Also, WinRT's cryptography components encrypt and decrypt blocks of data, and bitmap pixels are maintained in raw blocks of data too.

In the .NET Framework, the usual way to obtain a block of data is either with a byte array (`Byte[]`) or with a stream (such as when using the `MemoryStream` class). Of course, byte array and `MemoryStream` objects can't be passed to WinRT components directly. So, WinRT defines an `IBuffer` interface and objects that implement this interface represent raw blocks of data that can be passed to WinRT APIs. The WinRT `IBuffer` interface is defined as follows.

```
namespace Windows.Storage.Streams {
   public interface IBuffer {
      UInt32 Capacity { get; }       // Maximum size of the buffer (in bytes)
      UInt32 Length   { get; set; }  // Number of bytes currently in use by the buffer
   }
}
```

As you can see, an `IBuffer` object has a maximum size and length; oddly enough, this interface gives you no way to actually read from or write to the data in the buffer. The main reason for this is because WinRT types cannot express pointers in their metadata, because pointers do not map well to some languages (like JavaScript or safe C# code). So, an `IBuffer` object is really just a way to pass a memory address between the CLR and WinRT APIs. To access the bytes at the memory address, an internal COM interface, known as `IBufferByteAccess`, is used. Note that this interface is a COM interface (because it returns a pointer) and it is not a WinRT interface. The .NET Framework team has defined an internal RCW for this COM interface, which looks like the following.

```
namespace System.Runtime.InteropServices.WindowsRuntime {
   [Guid("905a0fef-bc53-11df-8c49-001e4fc686da")]
   [InterfaceType(ComInterfaceType.InterfaceIsIUnknown)]
   [ComImport]
   internal interface IBufferByteAccess {
      unsafe Byte* Buffer { get; }
   }
}
```

Internally, the CLR can take an `IBuffer` object, query for its `IBufferByteAccess` interface, and then query the `Buffer` property to get an unsafe pointer to the bytes contained within the buffer. With the pointer, the bytes can be accessed directly.

To avoid having developers write unsafe code that manipulates pointers, the Framework Class Library includes a `WindowsRuntimeBufferExtensions` class that defines a bunch of extension methods, which .NET Framework developers explicitly invoke to help pass blocks of data between CLR byte arrays and streams to WinRT `IBuffer` objects. To call these extension methods, make sure you add a `using System.Runtime.InteropServices.WindowsRuntime;` directive to your source code.

```
namespace System.Runtime.InteropServices.WindowsRuntime {
   public static class WindowsRuntimeBufferExtensions {
      public static IBuffer AsBuffer(this Byte[] source);
      public static IBuffer AsBuffer(this Byte[] source, Int32 offset, Int32 length);
      public static IBuffer AsBuffer(this Byte[] source, Int32 offset, Int32 length, Int32
capacity);

      public static IBuffer GetWindowsRuntimeBuffer(this MemoryStream stream);
      public static IBuffer GetWindowsRuntimeBuffer(this MemoryStream stream, Int32 position,
Int32 length);
   }
}
```

So, if you have a `Byte[]` and you want to pass it to a WinRT API requiring an `IBuffer`, you simply call `AsBuffer` on the `Byte[]` array. This effectively wraps the reference to the `Byte[]` inside an object that implements the `IBuffer` interface; the contents of the `Byte[]` array is not copied, so this is very efficient. Similarly, if you have a `MemoryStream` object wrapping a publicly visible `Byte[]` array buffer in it, you simply call `GetWindowsRuntimeBuffer` on it to wrap the reference to the `MemoryStream`'s buffer inside an object that implements the `IBuffer` interface. Again, the buffer's content is not copied, so this is very efficient. The following method demonstrates both scenarios.

```
private async Task ByteArrayAndStreamToIBuffer(IRandomAccessStream winRTStream, Int32 count) {
   Byte[] bytes = new Byte[count];
   await winRTStream.ReadAsync(bytes.AsBuffer(), (UInt32)bytes.Length, InputStreamOptions.None);
   Int32 sum = bytes.Sum(b => b);  // Access the bytes read via the Byte[]

   using (var ms = new MemoryStream())
   using (var sw = new StreamWriter(ms)) {
      sw.Write("This string represents data in a stream");
      sw.Flush();
      UInt32 bytesWritten = await winRTStream.WriteAsync(ms.GetWindowsRuntimeBuffer());
   }
}
```

WinRT's `IRandomAccessStream` interface implements WinRT's `IInputStream` interface defined as follows.

```
namespace Windows.Storage.Streams {
   public interface IOutputStream : IDisposable {
      IAsyncOperationWithProgress<UInt32, UInt32> WriteAsync(IBuffer buffer);
   }
}
```

When you call the AsBuffer or GetWindowsRuntimeBuffer extension methods in your code, these methods wrap the source object inside an object whose class implements the IBuffer interface. The CLR then creates a CCW for this object and passes it to the WinRT API. When the WinRT API queries the IBufferByteAccess interface's Buffer property to obtain a pointer to the underlying byte array, the byte array is pinned, and the address is returned to the WinRT API so it can access the data. The underlying byte array is unpinned when the WinRT API internally calls COM's Release method on the IBufferByteAccess interface.

If you call a WinRT API that returns an IBuffer back to you, then the data itself is probably in native memory, and you'll need a way to access this data from managed code. For this, we'll turn to some other extension methods defined by the WindowsRuntimeBufferExtensions class.

```
namespace System.Runtime.InteropServices.WindowsRuntime {
   public static class WindowsRuntimeBufferExtensions {
      public static Stream AsStream(this IBuffer source);
      public static Byte[] ToArray(this IBuffer source);
      public static Byte[] ToArray(this IBuffer source, UInt32 sourceIndex, Int32 count);

      // Not shown: CopyTo method to transfer bytes between an IBuffer and a Byte[]
      // Not shown: GetByte, IsSameData methods
   }
}
```

The AsStream method creates a Stream-derived object that wraps the source IBuffer. With this Stream object, you can access the data in the IBuffer by calling Stream's Read, Write, and similar methods. The ToArray method internally allocates a Byte[] and then copies all the bytes from the source IBuffer into the Byte[]; be aware that this extension method is potentially expensive in terms of memory consumption and CPU time.

The WindowsRuntimeBufferExtensions class also has several overloads of a CopyTo method that can copy bytes between an IBuffer and a Byte[]. It also has a GetByte method that retrieves a single byte at a time from an IBuffer and an IsSameData method that compares the contents of two IBuffer objects to see if their contents are identical. For most applications, it is unlikely that you will have a need to call any of these methods.

I'd also like to point out that the .NET Framework defines a System.Runtime.InteropServices.WindowsRuntimeBuffer class that allows you to create an IBuffer object whose bytes are in the managed heap. Similarly, there is a WinRT component called Windows.Storage.Streams.Buffer that allows you to create an IBuffer object whose bytes are in the native heap. For most .NET Framework developers, there should be no need to use either of these classes explicitly in your code.

Defining WinRT Components in C#

So far in this chapter, I've been focusing on how to consume WinRT components from C#. However, you can also define WinRT components in C# and then these components can be used by native C/C++, C#/Visual Basic, JavaScript, and potentially other languages too. Although this is possible to do, we need to think about the scenarios where this actually makes sense. For example, it makes no

sense at all to define a WinRT component in C# if the only consumers of the component are other managed languages that run on top of the CLR. This is because the WinRT type system has far fewer features, which make it much more restrictive than the CLR's type system.

I also don't think it makes a lot of sense to implement a WinRT component with C# that could be consumed by native C/C++ code. Developers using native C/C++ to implement their application are probably doing so because they are very concerned about performance and/or memory consumption. They are unlikely to want to take advantage of a WinRT component implemented with managed code, because this forces the CLR to load into their process and thus increases their memory requirements and performance due to the garbage collections and just-in-time compiling of code. For this reason, most WinRT components (like those that ship with Windows itself) are implemented in native code. Of course, there may be some parts of a native C++ app where performance is not so sensitive and, at these times, it may make sense to leverage .NET Framework functionality in order to improve productivity. For example, Bing Maps uses native C++ to draw its UI by using DirectX, but it also uses C# for its business logic.

So, it seems to me that the sweet spot for C#-implemented WinRT components is for Windows Store app developers who want to build their user interface with HTML and CSS and then use JavaScript as the glue code to tie the UI with business logic code that is implemented in a C# WinRT component. Another scenario would be to leverage existing Framework Class Library functionality (like Windows Communication Foundation) from an HTML/JavaScript app. Developers working with HTML and JavaScript are already willing to accept the kind of performance and memory consumption that comes with a browser engine and may be willing to even further accept the additional performance and memory consumption that comes along with also using the CLR.

To build a WinRT component with C#, you must first create a Microsoft Visual Studio "Windows Runtime Component" project. What this really does is create a normal class library project; however, the C# compiler will be spawned with the `/t:winmdobj` command line switch in order to produce a file with a .winmdobj file extension. With this switch specified, the compiler emits some IL code differently than it normally would. For example, WinRT components add and remove delegates to events differently than how the CLR does it, so the compiler emits different code for an event's `add` and `remove` methods when this compiler switch is specified. I'll show how to explicitly implement an event's `add` and `remove` methods later in this section.

After the compiler produces the .winmdobj file, the WinMD export utility (WinMDExp.exe) is spawned passing to it the .winmdobj, .pdb, and .xml (doc) files produced by the compiler. The WinMDExp.exe utility examines your file's metadata, ensuring that your types adhere to the various WinRT type system rules, as discussed at the beginning of this chapter. The utility also modifies the metadata contained in the .winmdobj file; it does not alter the IL code at all. Specifically, the utility maps any CLR types to the equivalent WinRT types. For example, references to the .NET Framework's `IList<String>` type are changed to WinRT's `IVector<String>` type. The output of the WinMDExp.exe utility is a .winmd file that other programming languages can consume.

You can use the .NET Framework's Intermediate Disassembler utility (ILDasm.exe) to inspect the contents of a .winmd file. By default, ILDasm.exe shows you the raw contents of the file. However,

ILDasm.exe supports a `/project` command-line switch that shows you what the metadata would look like after the CLR projected the WinRT types into their .NET Framework equivalents.

The following code demonstrates how to implement various WinRT components in C#. The components leverage many of the features discussed throughout this chapter, and there are a lot of comments throughout the code to explain what is going on. If you need to implement a WinRT component in C#, I'd suggest using the code I show here as a model.

Important When managed code consumes a WinRT component also written in managed code, the CLR treats the WinRT component as if it were a regular managed component. That is, the CLR will not create CCWs and RCWs, and therefore, it will not invoke the WinRT APIs via these wrappers. This improves performance greatly. However, when testing the component, APIs are not being invoked the way they would be if called from another language (like native C/C++ or JavaScript). So, aside from the performance and memory overhead not being real-life, managed code can pass `null` to a WinRT API requiring a `String` without an `ArgumentNullException` being thrown. And, WinRT APIs implemented in managed code can manipulate arrays that are passed in, and the caller will see the changed array contents when the API returns; normally, the WinRT type system forbids modifying arrays passed into an API. These are just some of the differences that you'll be able to observe, so beware.

```
/******************************************************************************
Module:  WinRTComponents.cs
Notices: Copyright (c) 2012 by Jeffrey Richter
******************************************************************************/

using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using System.Threading;
using System.Threading.Tasks;
using Windows.Foundation;
using Windows.Foundation.Metadata;

// The namespace MUST match the assembly name and cannot be "Windows"
namespace Wintellect.WinRTComponents {
   // [Flags]  // Must not be present if enum is int; required if enum is uint
   public enum WinRTEnum : int {    // Enums must be backed by int or uint
      None,
      NotNone
   }


   // Structures can only contain core data types, String, & other structures
   // No constructors or methods are allowed
   public struct WinRTStruct {
      public Int32 ANumber;
      public String AString;
```

```
   public WinRTEnum AEnum;    // Really just a 32-bit integer
}


// Delegates must have WinRT-compatible types in the signature (no BeginInvoke/EndInvoke)
public delegate String WinRTDelegate(Int32 x);


// Interfaces can have methods, properties, & events but cannot be generic.
public interface IWinRTInterface {
   // Nullable<T> marshals as IReference<T>
   Int32? InterfaceProperty { get; set; }
}

// Members without a [Version(#)] attribute default to the class's
// version (1) and are part of the same underlying COM interface
// produced by WinMDExp.exe.
[Version(1)]
// Class must be derived from Object, sealed, not generic,
// implement only WinRT interfaces, & public members must be WinRT types
public sealed class WinRTClass : IWinRTInterface {
   // Public fields are not allowed

   #region Class can expose static methods, properties, and events
   public static String StaticMethod(String s) { return "Returning " + s; }
   public static WinRTStruct StaticProperty { get; set; }

   // In JavaScript 'out' parameters are returned as objects with each
   // parameter becoming a property along with the return value
   public static String OutParameters(out WinRTStruct x, out Int32 year) {
      x = new WinRTStruct { AEnum = WinRTEnum.NotNone, ANumber = 333, AString = "Jeff" };
      year = DateTimeOffset.Now.Year;
      return "Grant";
   }
   #endregion

   // Constructor can take arguments but not out/ref arguments
   public WinRTClass(Int32? number) { InterfaceProperty = number; }

   public Int32? InterfaceProperty { get; set; }

   // Only ToString is allowed to be overridden
   public override String ToString() {
      return String.Format("InterfaceProperty={0}",
         InterfaceProperty.HasValue ? InterfaceProperty.Value.ToString() : "(not set)");
   }

   public void ThrowingMethod() {
      throw new InvalidOperationException("My exception message");

      // To throw a specific HRESULT, use COMException instead
      //const Int32 COR_E_INVALIDOPERATION = unchecked((Int32)0x80131509);
      //throw new COMException("Invalid Operation", COR_E_INVALIDOPERATION);
   }

   #region Arrays are passed, returned OR filled; never a combination
   public Int32 PassArray([ReadOnlyArray] /* [In] implied */ Int32[] data) {
```

```
   // NOTE: Modified array contents MAY not be marshaled out; do not modify the array
   return data.Sum();
}

public Int32 FillArray([WriteOnlyArray] /* [Out] implied */ Int32[] data) {
   // NOTE: Original array contents MAY not be marshaled in;
   // write to the array before reading from it
   for (Int32 n = 0; n < data.Length; n++) data[n] = n;
   return data.Length;
}

public Int32[] ReturnArray() {
   // Array is marshaled out upon return
   return new Int32[] { 1, 2, 3 };
}
#endregion

// Collections are passed by reference
public void PassAndModifyCollection(IDictionary<String, Object> collection) {
   collection["Key2"] = "Value2";  // Modifies collection in place via interop
}

#region Method overloading
// Overloads with same # of parameters are considered identical to JavaScript
public void SomeMethod(Int32 x) { }

[Windows.Foundation.Metadata.DefaultOverload]  // Makes this method the default overload
public void SomeMethod(String s) { }
#endregion

#region Automatically implemented event
public event WinRTDelegate AutoEvent;

public String RaiseAutoEvent(Int32 number) {
   WinRTDelegate d = AutoEvent;
   return (d == null) ? "No callbacks registered" : d(number);
}
#endregion

#region Manually implemented event
// Private field that keeps track of the event's registered delegates
private EventRegistrationTokenTable<WinRTDelegate> m_manualEvent = null;

// Manual implementation of the event's add and remove methods
public event WinRTDelegate ManualEvent {
   add {
      // Gets the existing table, or creates a new one if the table is not yet initialized
      return EventRegistrationTokenTable<WinRTDelegate>
         .GetOrCreateEventRegistrationTokenTable(ref m_manualEvent)
         .AddEventHandler(value);
   }
```

```
            remove {
               EventRegistrationTokenTable<WinRTDelegate>
                  .GetOrCreateEventRegistrationTokenTable(ref m_manualEvent)
                  .RemoveEventHandler(value);
            }
         }

         public String RaiseManualEvent(Int32 number) {
            WinRTDelegate d = EventRegistrationTokenTable<WinRTDelegate>
               .GetOrCreateEventRegistrationTokenTable(ref m_manualEvent).InvocationList;
            return (d == null) ? "No callbacks registered" : d(number);
         }
         #endregion

         #region Asynchronous methods
         // Async methods MUST return IAsync[Action|Operation](WithProgress)
         // NOTE: Other languages see the DataTimeOffset as Windows.Foundation.DateTime
         public IAsyncOperationWithProgress<DateTimeOffset, Int32> DoSomethingAsync() {
            // Use the System.Runtime.InteropServices.WindowsRuntime.AsyncInfo's Run methods to
            // invoke a private method written entirely in managed code
            return AsyncInfo.Run<DateTimeOffset, Int32>(DoSomethingAsyncInternal);
         }

         // Implement the async operation via a private method using normal .NET technologies
         private async Task<DateTimeOffset> DoSomethingAsyncInternal(
            CancellationToken ct, IProgress<Int32> progress) {

            for (Int32 x = 0; x < 10; x++) {
               // This code supports cancellation and progress reporting
               ct.ThrowIfCancellationRequested();
               if (progress != null) progress.Report(x * 10);
               await Task.Delay(1000); // Simulate doing something asynchronously
            }
            return DateTimeOffset.Now; // Ultimate return value
         }

         public IAsyncOperation<DateTimeOffset> DoSomethingAsync2() {
            // If you don't need cancellation & progress, use
            // System.WindowsRuntimeSystemExtensions' AsAsync[Action|Operation] Task
            // extension methods (these call AsyncInfo.Run internally)
            return DoSomethingAsyncInternal(default(CancellationToken), null).AsAsyncOperation();
         }
         #endregion

         // After you ship a version, mark new members with a [Version(#)] attribute
         // so that WinMDExp.exe puts the new members in a different underlying COM
         // interface. This is required since COM interfaces are supposed to be immutable.
         [Version(2)]
         public void NewMethodAddedInV2() {}
      }
   }
```

The following JavaScript code demonstrates how to access all of the previous WinRT components and features.

```
function () {
   // Make accessing the namespace more convenient in the code
   var WinRTComps = Wintellect.WinRTComponents;

   // NOTE: The JavaScript VM projects WinRT APIs via camel casing

   // Access WinRT type's static method & property
   var s = WinRTComps.WinRTClass.staticMethod(null); // NOTE: JavaScript pass "null" here!
   var struct = { anumber: 123, astring: "Jeff", aenum: WinRTComps.WinRTEnum.notNone };
   WinRTComps.WinRTClass.staticProperty = struct;
   s = WinRTComps.WinRTClass.staticProperty;  // Read it back

   // If the method has out parameters, they and the return value
   // are returned as an object's properties
   var s = WinRTComps.WinRTClass.outParameters();
   var name = s.value; // Return value
   var struct = s.x;   // an 'out' parameter
   var year = s.year;  // another 'out' parameter

   // Construct an instance of the WinRT component
   var winRTClass = new WinRTComps.WinRTClass(null);
   s = winRTClass.toString();  // Call ToString()

   // Demonstrate throw and catch
   try { winRTClass.throwingMethod(); }
   catch (err) { }

   // Array passing
   var a = [1, 2, 3, 4, 5];
   var sum = winRTClass.passArray(a);

   // Array filling
   var arrayOut = [7, 7, 7];   // NOTE: fillArray sees all zeros!
   var length = winRTClass.fillArray(arrayOut); // On return, arrayOut = [0, 1, 2]

   // Array returning
   a = winRTClass.returnArray();   // a = [ 1, 2, 3]

   // Pass a collection and have its elements modified
   var localSettings = Windows.Storage.ApplicationData.current.localSettings;
   localSettings.values["Key1"] = "Value1";
   winRTClass.passAndModifyCollection(localSettings.values);
   // On return, localSettings.values has 2 key/value pairs in it

   // Call overloaded method
   winRTClass.someMethod(5);       // Actually calls SomeMethod(String) passing "5"

   // Consume the automatically implemented event
   var f = function (v) { return v.target; };
   winRTClass.addEventListener("autoevent", f, false);
   s = winRTClass.raiseAutoEvent(7);
```

```
   // Consume the manually implemented event
   winRTClass.addEventListener("manualevent", f, false);
   s = winRTClass.raiseManualEvent(8);

   // Invoke asynchronous method supporting progress, cancelation, & error handling
   var promise = winRTClass.doSomethingAsync();
   promise.then(
      function (result) { console.log("Async op complete: " + result); },
      function (error) { console.log("Async op error: " + error); },
      function (progress) {
         console.log("Async op progress: " + progress);
         //if (progress == 30) promise.cancel();   // To test cancelation
      });
}
```