C# 10 in a Nutshell Supplement

Chapter 26

Serialization

This chapter introduces serialization and deserialization, the mechanism by which objects can be represented in a flat-text or binary form. Unless otherwise stated, the types in this chapter all exist in the following namespaces:

```
System.Runtime.Serialization
System.Xml.Serialization
System.Text.Json
```

Serialization Concepts

Serialization is the act of taking an in-memory object or *object graph* (set of objects that reference one another) and flattening it into a stream of bytes, XML, JSON, or a similar representation that can be stored or transmitted. *Deserialization* works in reverse, taking a data stream and reconstituting it into an in-memory object or object graph.

Serialization and deserialization are typically used to do the following:

- Transmit objects across a network or application boundary
- Store representations of objects within a file or database

Another, less common use is to deep-clone objects. You also can use the data contract and XML serialization engines as general-purpose tools for loading and saving XML files of a known structure, whereas the JSON serializer can do the same for JSON files.

.NET supports serialization and deserialization both from the perspective of clients wanting to serialize and deserialize objects, and from the perspective of types wanting some control over how they are serialized.

Serialization Engines

There are four serialization engines in .NET 5+ and .NET Core:

- `XmlSerializer` (XML)
- `JsonSerializer` (JSON)
- The (somewhat redundant) data contract serializer (XML and JSON)
- The binary serializer (binary)

> The binary serializer is being phased out due to security vulnerabilities. For details, see https://aka.ms/binaryformatter

If you're serializing to XML, you can choose between `XmlSerializer` and the data contract serializer. `XmlSerializer` offers greater flexibility on how the XML is structured, whereas the data contract serializer has the ability to preserve shared object references.

If you're serializing to JSON, you also have a choice. `JsonSerializer` offers the best performance, and (as of .NET 5), there are few reasons to favor the data contract serializer. And if `JsonSerializer` doesn't provide the features you need, another choice is the popular third-party Json.NET library.

If you need to interoperate with legacy SOAP-based web services, the data contract serializer is the best choice.

And if you don't care about the format, the binary serialization engine is the most powerful and easiest to use. The output, however, is not human-readable and it's less version-tolerant than the other serializers.

> Given that the binary serializer is being phased out, `JsonSerializer` is now best choice, if you don't care about the format.

The following tables compares each of the engines. More stars equate to a better score.

Feature	XmlSerializer	JsonSerializer	Data contract serializer	Binary serializer
Level of automation	****	*****	***	*****
Output	XML	JSON	XML or JSON	Binary
Type coupling	Loose	Loose	Loose	Tight
Version tolerance	*****	*****	*****	***
Can deserialize subtypes	With help	No	With help	Yes
Preserves object references	No	From .NET 5	With XML	Yes
Can serialize nonpublic fields	No	From .NET 5	Yes	Yes
Suitable for interoperable messaging	Yes	Yes	Yes	No
Flexibility in output format	****	***	**	-
Compact output	**	***	**	****
Performance	* to ***	****	***	***

Note that the XML serialization engine requires that you recycle the same `XmlSerializer` object for good performance.

Why four engines?

The reason for there being four engines is partly historical. The .NET Framework originally started out with two distinct goals in serialization:

- Serializing .NET object graphs with full type and reference fidelity
- Interoperating with XML and SOAP messaging standards

The first led to the binary serializer (which was used by .NET Remoting); the second led to the `XmlSerializer` (which was used by ASMX web services).

With the release of Windows Communication Foundation (WCF) in 2006, a new serialization engine was required—the *data contract serializer*—and it was hoped that the new engine could largely replace the older two.

However, because its design focused heavily on features relevant to interoperable messaging, it never fully achieved this goal, and the two older engines remained useful.

WCF was designed to be format-neutral, but in practice it was shaped by needs of complex SOAP protocols, which later lost popularity in favor of REST and JSON. This led, at first, to Microsoft adding JSON support to the data contract serializer, but eventually to the demise of WCF and its exclusion from .NET Core 3. The data contract serializer remains in .NET Core and .NET 5+, although the exclusion of WCF has diminished its role, as has Microsoft's addition of `JsonSerializer` to .NET Core 3.

XmlSerializer

The XML serialization engine can produce only XML, and it is less powerful than the binary and data contract serializers in saving and restoring a complex object graph (it cannot restore shared object references). It's the most flexible of the four, however, in following an arbitrary output structure. For instance, you can choose whether properties are serialized to elements or attributes and the handling of a collection's outer element. The XML engine also provides excellent version tolerance. `XmlSerializer` was used by the legacy ASMX web services.

JsonSerializer

The JSON serializer is fast and efficient. It also offers good version tolerance and allows the use of custom converters for flexibility. `JsonSerializer` has been used by ASP.NET since ASP.NET Core 3 (removing the prior dependency on Json.NET), though you can opt back in to Json.NET should its features be required.

From .NET 5, the JSON serializer can preserve object references.

The data contract serializer

The data contract serializer supports a *data contract* model that helps you decouple the low-level details of the types you want to serialize from the structure of the serialized data. This provides excellent version tolerance, meaning you can deserialize data that was serialized from an earlier or later version of a type. You can even deserialize types that have been renamed or moved to a different assembly.

The data contract serializer can cope with most object graphs, although it can require more assistance than the binary serializer. You also can use it as a general-purpose tool for reading/writing XML files, if you're flexible on how the XML is structured. (If you need to store data in attributes or cope with XML elements presenting in an arbitrary order, you cannot use the data contract serializer.)

The binary serializer

The binary serialization engine is easy to use, highly automatic, and well supported throughout .NET 5+ and .NET Core 3 (and even more so in .NET Framework). Quite often, a single attribute is all that's required to make a complex type fully serializable. The binary serializer is also faster than the data contract serializer when full type fidelity is needed. However, it tightly couples a type's internal structure to the format of the serialized data, resulting in poor version tolerance (although it can tolerate the simple addition of a field). The binary engine emits only binary data; it cannot produce XML or JSON in .NET Core and .NET 5+. (In .NET Framework, there's a formatter for SOAP-based messaging that provides limited XML support.)

The IXmlSerializable hook

For complex XML serialization tasks, you can implement `IXmlSerializable` and do the serialization yourself with an `XmlReader` and `XmlWriter`. The `IXmlSerializable` interface is recognized both by `XmlSerializer` and by the data contract serializer, so you can use it selectively to handle the more complicated types. We describe `XmlReader` and `XmlWriter` in detail in Chapter 11.

Formatters

The output of the data contract and binary serializers is shaped by a pluggable *formatter*. The role of a formatter is the same with both serialization engines, although they use completely different classes to do the job.

A formatter shapes the final presentation to suit a particular medium or context of serialization. In .NET 5+ and .NET Core, the data contract serializer lets you choose between XML and JSON formatters, and in .NET

Framework you can also choose a binary formatter. A binary formatter is designed to work in a context for which an arbitrary stream of bytes will do—typically a file/stream or proprietary messaging packet. Binary output is usually smaller than XML or JSON.

The binary serializer offers only a binary formatter in .NET 5+ and .NET Core (in .NET Framework, there's also a SOAP formatter for XML-based messaging).

Explicit versus Implicit Serialization

Serialization and deserialization can be initiated in two ways.

The first is *explicitly*, by requesting that a particular object be serialized or deserialized. When you serialize or deserialize explicitly, you choose both the serialization engine and the formatter.

In contrast, *implicit* serialization is initiated by .NET. This happens when:

- A serializer recursively serializes a child object.
- You use a feature that relies on serialization, such as Web API.

Web API can work with either XML or JSON serialization.

Implicit serialization is less prevalent in .NET 5+ and .NET Core than in .NET Framework, which includes WCF (implicitly using the data contract serializer), Remoting (implicitly using the binary serialization engine), and ASMX Web Services (implicitly using `XmlSerializer`).

The XML Serializer

The `XmlSerializer` class in the `System.Xml.Serialization` namespace serializes and deserializes based on attributes in your classes.

Getting Started with Attribute-Based Serialization

To use `XmlSerializer`, you instantiate it and call `Serialize` or `Deserialize` with a `Stream` and object instance. To illustrate, suppose we define the following class:

```
public class Person
{
  public string Name;
  public int Age;
}
```

The following saves a `Person` to an XML file and then restores it:

```
Person p = new Person();
p.Name = "Stacey"; p.Age = 30;

var xs = new XmlSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  xs.Serialize (s, p);

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) xs.Deserialize (s);

Console.WriteLine (p2.Name + " " + p2.Age);   // Stacey 30
```

`Serialize` and `Deserialize` can work with a `Stream`, `XmlWriter`/`XmlReader`, or `TextWriter`/`TextReader`. Here's the resultant XML:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Stacey</Name>
  <Age>30</Age>
</Person>
```

`XmlSerializer` can serialize types without any attributes—such as our `Person` type. By default, it serializes all *public fields and properties* on a type. You can exclude members that you don't want serialized by applying the `XmlIgnore` attribute:

```
public class Person
{
  ...
  [XmlIgnore] public DateTime DateOfBirth;
}
```

`XmlSerializer` relies on a parameterless constructor for deserialization, throwing an exception if one is not present. (In our example, `Person` has an *implicit* parameterless constructor.) This also means that field initializers execute prior to deserialization:

```
public class Person
{
  public bool Valid = true;    // Executes before deserialization
}
```

Although `XmlSerializer` can serialize almost any type, it recognizes the following types and treats them specially:

- The primitive types, `DateTime`, `TimeSpan`, `Guid`, and nullable versions
- `byte[]` (which is converted to base 64)
- An `XmlAttribute` or `XmlElement` (whose contents are injected into the stream)
- Any type implementing `IXmlSerializable`
- Any collection type

The deserializer is version tolerant: it doesn't complain if elements or attributes are missing or if superfluous data is present.

Attributes, names, and namespaces

By default, fields and properties serialize to an XML element. You can request an XML attribute be used instead as follows:

```
[XmlAttribute] public int Age;
```

You can control an element or attribute's name as follows:

```
public class Person
{
  [XmlElement ("FirstName")] public string Name;
  [XmlAttribute ("RoughAge")] public int Age;
}
```

Here's the result:

```
<Person RoughAge="30" ...>
  <FirstName>Stacey</FirstName>
</Person>
```

The default XML namespace is blank. To specify an XML namespace, `[XmlElement]` and `[XmlAttribute]` both accept a `Namespace` argument. You can also assign a name and namespace to the type itself with `[XmlRoot]`:

```
[XmlRoot ("Candidate", Namespace = "http://mynamespace/test/")]
public class Person { ... }
```

This names the `person` element "Candidate" as well as assigning a namespace to this element and its children.

XML element order

`XmlSerializer` writes elements in the order in which they're defined in the class. You can change this by specifying an `Order` in the `XmlElement` attribute:

```
public class Person
{
  [XmlElement (Order = 2)] public string Name;
  [XmlElement (Order = 1)] public int Age;
}
```

If you use `Order` at all, you must use it throughout.

The deserializer is not fussy about the order of elements—they can appear in any sequence and the type will properly deserialize.

Subclasses and Child Objects

Subclassing the root type

Suppose that your root type has two subclasses, as follows:

```
public class Person { public string Name; }

public class Student : Person { }
public class Teacher : Person { }
```

and you want to write a reusable method to serialize the root type:

```
public void SerializePerson (Person p, string path)
{
  XmlSerializer xs = new XmlSerializer (typeof (Person));
  using (Stream s = File.Create (path))
    xs.Serialize (s, p);
}
```

To make this method work with a `Student` or `Teacher`, you must inform `XmlSerializer` about the subclasses. There are two ways to do this. The first is to register each subclass by applying the `XmlInclude` attribute:

```
[XmlInclude (typeof (Student))]
[XmlInclude (typeof (Teacher))]
public class Person { public string Name; }
```

The second is to specify each of the subtypes when constructing `XmlSerializer`:

```
XmlSerializer xs = new XmlSerializer (typeof (Person),
                       new Type[] { typeof (Student), typeof (Teacher) } );
```

In either case, the serializer responds by recording the subtype in the `type` attribute:

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="Student">
  <Name>Stacey</Name>
</Person>
```

This deserializer then knows from this attribute to instantiate a `Student` and not a `Person`.

> You can control the name that appears in the XML `type` attribute by applying `[XmlType]` to the subclass:
>
> ```
> [XmlType ("Candidate")]
> public class Student : Person { }
> ```
>
> Here's the result:

```
<Person xmlns:xsi="..."
        xsi:type="Candidate">
```

Serializing child objects

`XmlSerializer` automatically recurses object references such as the `HomeAddress` field in `Person`:

```
public class Person
{
  public string Name;
  public Address HomeAddress = new Address();
}

public class Address { public string Street, PostCode; }
```

To demonstrate:

```
Person p = new Person { Name = "Stacey" };
p.HomeAddress.Street = "Odo St";
p.HomeAddress.PostCode = "6020";
```

Here's the XML to which this serializes:

```
<Person ... >
  <Name>Stacey</Name>
  <HomeAddress>
    <Street>Odo St</Street>
    <PostCode>6020</PostCode>
  </HomeAddress>
</Person>
```

> If you have two fields or properties that refer to the same object, that object is serialized twice. If you need to preserve referential equality, you must use another serialization engine.

Subclassing child objects

Suppose that you need to serialize a `Person` that can reference *subclasses* of `Address`, as follows:

```
public class Address { public string Street, PostCode; }
public class USAddress : Address {  }
public class AUAddress : Address {  }

public class Person
{
  public string Name;
  public Address HomeAddress = new USAddress();
}
```

There are two distinct ways to proceed, depending on how you want the XML structured. If you want the element name always to match the field or property name with the subtype recorded in a `type` attribute

```
<Person ...>
  ...
  <HomeAddress xsi:type="USAddress">
    ...
  </HomeAddress>
</Person>
```

you use `[XmlInclude]` to register each of the subclasses with `Address`, as follows:

```
[XmlInclude (typeof (AUAddress))]
[XmlInclude (typeof (USAddress))]
public class Address
```

```
{
  public string Street, PostCode;
}
```

If, on the other hand, you want the element name to reflect the name of the subtype, to the following effect:

```
<Person ...>
  ...
  <USAddress>
    ...
  </USAddress>
</Person>
```

you instead stack multiple `[XmlElement]` attributes onto the field or property in the parent type:

```
public class Person
{
  public string Name;

  [XmlElement ("Address", typeof (Address))]
  [XmlElement ("AUAddress", typeof (AUAddress))]
  [XmlElement ("USAddress", typeof (USAddress))]
  public Address HomeAddress = new USAddress();
}
```

Each `XmlElement` maps an element name to a type. If you take this approach, you don't require the `[XmlInclude]` attributes on the `Address` type (although their presence doesn't break serialization).

> If you omit the element name in `[XmlElement]` (and specify just a type), the type's default name is used (which is influenced by `[XmlType]` but not `[XmlRoot]`).

Serializing Collections

`XmlSerializer` recognizes and serializes concrete collection types without intervention:

```
public class Person
{
  public string Name;
  public List<Address> Addresses = new List<Address>();
}

public class Address { public string Street, PostCode; }
```

Here's the XML to which this serializes:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    ...
  </Addresses>
</Person>
```

The `[XmlArray]` attribute lets you rename the *outer* element (i.e., `Addresses`).

The `[XmlArrayItem]` attribute lets you rename the *inner* elements (i.e., the `Address` elements).

For instance, the following class

```
public class Person
{
  public string Name;

  [XmlArray ("PreviousAddresses")]
  [XmlArrayItem ("Location")]
  public List<Address> Addresses = new List<Address>();
}
```

serializes to this:

```
<Person ... >
  <Name>...</Name>
  <PreviousAddresses>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    ...
  </PreviousAddresses>
</Person>
```

The `XmlArray` and `XmlArrayItem` attributes also allow you to specify XML namespaces.

To serialize collections *without* the outer element, for example

```
<Person ... >
  <Name>...</Name>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
</Person>
```

instead add `[XmlElement]` to the collection field or property:

```
public class Person
{
  ...
  [XmlElement ("Address")]
  public List<Address> Addresses = new List<Address>();
}
```

Working with subclassed collection elements

The rules for subclassing collection elements follow naturally from the other subclassing rules. To encode subclassed elements with the `type` attribute, for example

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address xsi:type="AUAddress">
```

```
    ...
```

add `[XmlInclude]` attributes to the base (`Address`) type, as we did earlier. This works whether or not you suppress serialization of the outer element.

If you want subclassed elements to be named according to their type, for example

```
<Person ... >
  <Name>...</Name>
  <!—start of optional outer element—>
  <AUAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </AUAddress>
  <USAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </USAddress>
  <!—end of optional outer element—>
</Person>
```

you must stack multiple `[XmlArrayItem]` or `[XmlElement]` attributes onto the collection field or property.

Stack multiple `[XmlArrayItem]` attributes if you want to *include* the outer collection element:

```
[XmlArrayItem ("Address",   typeof (Address))]
[XmlArrayItem ("AUAddress", typeof (AUAddress))]
[XmlArrayItem ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

Stack multiple `[XmlElement]` attributes if you want to *exclude* the outer collection element:

```
[XmlElement ("Address",   typeof (Address))]
[XmlElement ("AUAddress", typeof (AUAddress))]
[XmlElement ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

IXmlSerializable

Although attribute-based XML serialization is flexible, it has limitations. For instance, you cannot add serialization hooks—nor can you serialize nonpublic members. It's also awkward to use if the XML might present the same element or attribute in a number of different ways.

On that last issue, you can push the boundaries somewhat by passing an `XmlAttributeOverrides` object into `XmlSerializer`'s constructor. There comes a point, however, when it's easier to take an imperative approach. This is the job of `IXmlSerializable`:

```
public interface IXmlSerializable
{
  XmlSchema GetSchema();
  void ReadXml (XmlReader reader);
  void WriteXml (XmlWriter writer);
}
```

Implementing this interface gives you total control over the XML that's read or written.

> A collection class that implements `IXmlSerializable` bypasses `XmlSerializer`'s rules for serializing collections. This can be useful if you need to serialize a collection with a payload—in other words, additional fields or properties that would otherwise be ignored.

The rules for implementing `IXmlSerializable` are as follows:

- `ReadXml` should read the outer start element, then the content, and then the outer end element.

- `WriteXml` should write just the content.

Here's an example:

```
using System;
using System.Xml;
using System.Xml.Schema;
using System.Xml.Serialization;

public class Address : IXmlSerializable
{
  public string Street, PostCode;

  public XmlSchema GetSchema() { return null; }

  public void ReadXml(XmlReader reader)
  {
    reader.ReadStartElement();
    Street   = reader.ReadElementContentAsString ("Street", "");
    PostCode = reader.ReadElementContentAsString ("PostCode", "");
    reader.ReadEndElement();
  }

  public void WriteXml (XmlWriter writer)
  {
    writer.WriteElementString ("Street", Street);
    writer.WriteElementString ("PostCode", PostCode);
  }
}
```

Serializing and deserializing an instance of `Address` via `XmlSerializer` automatically calls the `WriteXml` and `ReadXml` methods. Further, if `Person` were defined like this

```
public class Person
{
  public string Name;
  public Address HomeAddress;
}
```

`IXmlSerializable` would be called upon selectively to serialize the `HomeAddress` field.

We describe `XmlReader` and `XmlWriter` at length in the first section of Chapter 11. Also in Chapter 11, in "Patterns for Using XmlReader/XmlWriter," we provide examples of `IXmlSerializable`-ready classes.

The JSON Serializer

`JsonSerializer` (in the `System.Text.Json` namespace) is straightforward to use because of the simplicity of the JSON format. The root of a JSON document is either an array or an object. Under that root are properties, which can be an object, array, string, number, `"true"`, `"false"`, or `"null"`. The JSON serializer directly maps class property names to property names in JSON.

Getting Started

Assuming `Person` is defined like this

```
public class Person
{
  public string Name { get; set; }
}
```

we can serialize it to a JSON string by calling `JsonSerializer.Serialize`:

```
var p = new Person { Name = "Ian" };
string json = JsonSerializer.Serialize (p,
                new JsonSerializerOptions { WriteIndented = true });
```

Here is the result:

```
{
  Name: "Ian"
}
```

The `JsonSerializer.Deserialize` method does the reverse, and deserializes:

```
Person p2 = JsonSerializer.Deserialize<Person> (json);
```

Deserializing Immutable Types

The deserializer can populate types with read-only properties—as long as a public constructor is present with parameter names that (case-insensitively) match the properties being deserialized:

```
var p = new Person ("Joe", "Bloggs");
string json = JsonSerializer.Serialize (p);
Person p2 = JsonSerializer.Deserialize<Person> (json);

public class Person
{
  public string FirstName { get; }
  public string LastName { get; }

  public Person (string firstName, string lastName)
    => (FirstName, LastName) = (firstName, lastName);
}
```

If a type defines more than one constructor with parameters, you must tell the deserializer which one to use by applying the `[JsonConstructor]` attribute to the correct constructor.

Records (introduced in C# 9) work well with the deserializer by default.

Serializing Child Objects

Suppose that we define `Person` to have a home and work `Address`:

```
public class Address
{
  public string Street { get; set; }
  public string PostCode { get; set; }
}

public class Person
{
  public string Name { get; set; }
  public Address HomeAddress { get; set; }
  public Address WorkAddress { get; set; }
}
```

We can serialize this with no extra work:

```
var home = new Address { Street = "1 Main St.", PostCode="11235" };
var work = new Address { Street = "4 Elm Ln.", PostCode="31415" };
var p = new Person { Name = "Ian", HomeAddress = home, WorkAddress = work };

Console.WriteLine (JsonSerializer.Serialize (p,
```

```
                new JsonSerializerOptions { WriteIndented = true } ));
```

Upon encountering `HomeAddress` and `WorkAddress`, the serializer creates JSON objects:

```
{
  "Name": "Ian",
  "HomeAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "Street": "4 Elm Ln.",
    "PostCode": "31415"
  }
}
```

Note, though, what happens when we set `HomeAddress` and `WorkAddress` to the same object instance:

```
var p = new Person { Name = "Ian", HomeAddress = home, WorkAddress = home };
```

Here's the output:

```
{
  "Name": "Ian",
  "HomeAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  }
}
```

There is no information in the JSON to indicate that `HomeAddress` and `WorkAddress` were originally the same object instance. When deserialized, two separate instances of `Address` will be created and assigned to the respective properties.

Preserving object references

You can solve this problem (from .NET 5) by asking the serializer to preserve object references using the following option:

```
new JsonSerializerOptions
{
  ReferenceHandler = ReferenceHandler.Preserve,
  WriteIndented = true
}
```

The output now looks like this:

```
{
  "$id": "1",
  "Name": "Ian",
  "HomeAddress": {
    "$id": "2",
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "$ref": "2"
  }
}
```

This will deserialize without duplicating the address object—as long as you provide the same option to the deserializer.

Preserving object references also allows the `JsonSerializer` to handle cycles in the object graph.

Serializing Collections

`JsonSerializer` automatically serializes collections. Collections can appear in an object's properties as well as in the root object itself. We can illustrate the latter by using the `Person` and `Address` classes that we defined at the beginning of the preceding section:

```
var sara = new Person { Name = "Sara" };
var ian = new Person { Name = "Ian" };

Console.WriteLine (JsonSerializer.Serialize (new[] { sara, ian },
  new JsonSerializerOptions { WriteIndented = true }));
```

Here's the result:

```
[
  {
    "Name": "Sara"
  },
  {
    "Name": "Ian"
  }
]
```

The following deserializes the JSON:

```
Person[] people = JsonSerializer.Deserialize<Person[]> (json);
```

It is possible to serialize a collection containing differently typed objects:

```
var sara = new Person { Name = "Sara" };
var addr = new Address { Street = "1 Main St.", PostCode = "11235" };

Console.WriteLine (JsonSerializer.Serialize (new object[] { sara, addr },
  new JsonSerializerOptions { WriteIndented = true }));
```

This yields the following:

```
[
  {
    "Name": "Sara"
  },
  {
    "Street": "1 Main St.",
    "PostCode": "11235"
  }
]
```

Deserializing such collections is clumsy because the type of each element is not written into the JSON. You need to take the low-level approach of deserializing to `JsonElement[]` and then enumerating each property:

```
var deserialized = JsonSerializer.Deserialize<JsonElement[]>(json);
foreach (var element in deserialized)
{
  foreach (var prop in element.EnumerateObject())
    Console.WriteLine ($"{prop.Name}: {prop.Value}");
  Console.WriteLine ("---");
}

// Output:
```

```
Name: Sara
---
Street: 1 Main St.
PostCode: 11235
```

We describe how to use `JsonElement` in "JsonDocument" in Chapter 11.

Controlling Serialization with Attributes

You can control the serialization process with attributes defined in the `System.Text.Json.Serialization` namespace.

JsonIncludeAttribute (from .NET 5)

By default, the serializer/deserializer ignores fields unless you apply the `[JsonInclude]` attribute:

```
class Person
{
  [JsonInclude]
  public string Phone;   // This will now be included
}
```

This attribute can also be applied to properties with a non-public `set` accessor to instruct the deserializer to invoke it via reflection:

```
class Person
{
  [JsonInclude]
  public string Phone { get; private set; }
}
```

JsonIgnoreAttribute

By default, the JSON serializer serializes all properties unless you opt out by applying the `JsonIgnore` attribute:

```
public class Person
{
  public string Name { get; set; }

  [JsonIgnore]
  public decimal NetWorth { get; set; }   // Not serialized
}
```

JsonPropertyNameAttribute

If the JSON property name differs from the C# property name, you can create a mapping with `[JsonPropertyName]`. For example, if the JSON property name is `"FullName"`, and the C# property name is `Name`, we could create a mapping, as follows:

```
public class Person
{
  [JsonPropertyName("FullName")]
  public string Name { get; set; }
}
```

This serializes to the following:

```
{
  "FullName":"...",
}
```

JsonExtensionDataAttribute

Consider a web API that returns instances of a `Person` class and a client that uses the API. Both are maintained by different organizations. If the API author adds a new property to the `Person` class (such as `Age`), the client is still able to deserialize the JSON with its old `Person` class, because it will simply skip over the unknown `Age` property. However, suppose that the client then updates its instance of `Person`, serializes it, and sends it back to the API. The original `Age` value is then lost.

To illustrate, we'll have the web API define `Person` as

```
public class Person_// v2
{
  public int Id { get; set; }
  public string Name { get; set; }
  public int Age { get; set; }     // New property
}
```

which would generate JSON like this:

```
{
  "Id": 27182,
  "Name": "Sara",
  "Age": 35
}
```

If we deserialize that JSON into an older version of the class (without the `Age` property)

```
public class Person_// v1
{
  public int Id { get; set; }
  public string Name { get; set; }
}
```

the age information has no place to go.

If we later serialize our version and send it back to the API, our JSON will not contain an `Age` property, and the API will interpret `Age` to be zero (the default value for an integer).

`JsonExtensionDataAttribute` solves that problem by providing a mechanism to store all unrecognized properties so that their values can be used when reserializing. When the attribute is placed on a property of type `IDictionary<string,TValue>` (`TValue` must be `object` or `JsonElement`), the serializer uses that property to persist the unrecognized JSON properties; no information is lost:

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }

  [JsonExtensionData]
  public IDictionary<string, JsonElement> Storage { get; set; } =
    new Dictionary<string, JsonElement>();
}
```

JsonConverterAttribute

This attribute is used to specify a type used to convert data to and from JSON. We discuss this further in the next section.

Customizing Data Conversion

Suppose that you need to interoperate with an API provider that encodes dates with the Unix timestamp format (number of seconds since 1/1/1970):

```
{
  "Id":27182,
  "Name":"Sara",
  "Born":464572800   // Number of seconds since 1/1/1970
}
```

We would like to deserialize this into a class that uses the .NET `DateTime` class:

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public DateTime Born { get; set; }
}
```

We can achieve this by writing a custom data converter:

```
public class UnixTimestampConverter : JsonConverter<DateTime>
{
  public override DateTime Read (ref Utf8JsonReader reader, Type type,
                                 JsonSerializerOptions options)
  {
    if (reader.TryGetInt32(out int timestamp))
      return new DateTime (1970, 1, 1).AddSeconds (timestamp);

    throw new Exception ("Expected the timestamp as a number.");
  }

  public override void Write (Utf8JsonWriter writer, DateTime value,
                              JsonSerializerOptions options)
  {
    int timestamp = (int)(value - new DateTime(1970, 1, 1)).TotalSeconds;
    writer.WriteNumberValue(timestamp);
  }
}
```

Then we can either apply the `[JsonConverter]` to the properties that we want to convert

```
[JsonConverter(typeof(UnixTimestampConverter))]
public DateTime Born { get; set; }
```

or, if the API is consistent in its representation of data types, make the converter act as a default:

```
JsonSerializerOptions opts = new JsonSerializerOptions();
opts.Converters.Add (new UnixTimestampConverter());
var sara = JsonSerializer.Deserialize<Person> (json, opts);
```

The latter instructs the serializer to use `UnixTimestampConverter` every time it encounters a `DateTime`.

Handling null values (from .NET 5)

As a performance optimization, the serializer and deserializer normally bypasses custom converters when it encounters null values. If you want a custom converter to handle nulls, you must override the `HandleNull` property as follows:

```
public class UnixTimestampConverter : JsonConverter<DateTime?>
{
  public override bool HandleNull => true;
  ...
```

This works both with reference-type nulls (e.g., strings) and nullable value types (as in our example).

JSON Serialization Options

The serializer accepts an optional `JsonSerializationOptions` parameter, allowing additional control over the serialization and deserialization process. The following subsections present the most useful options.

> From .NET 5, the default serialization options differ depending on whether or not you're writing a web app.

Here are the options that differ depending on the kind of application that you're writing:

Option	Normal default	Web app default
`PropertyNameCaseInsensitive`	False	True
`PropertyNamingPolicy`	(none)	`CamelCase`
`NumberHandling`	`Strict`	`AllowReadingFromString`

Regardless of the kind of application you're writing, you can access each set of default settings via the following static properties:

```
JsonSerializerDefaults.Default
JsonSerializerDefaults.Web
```

You can clone either of these by using the constructor that accepts another `JsonSerializationOptions`:

```
var options = new JsonSerializerOptions (JsonSerializerDefaults.Web);
```

WriteIndented

We have set `WriteIndented` to true throughout this section to instruct the serializer to emit whitespace to generate more human-readable JSON. The default is false, which results in everything being crammed onto one line.

AllowTrailingCommas

The JSON spec requires properties and array elements to be comma separated but does not allow trailing commas:

```
{
  "Name":"Dylan",
  "LuckyNumbers": [10, 7, ],
  "Age":46,
}
```

The trailing commas after 7 and 46 are not allowed by default. To enable them, do this:

```
  var commaTolerant = JsonSerializer.Deserialize<Person> (brokenJson,
    new JsonSerializerOptions { AllowTrailingCommas = true });
```

ReadCommentHandling

By default, the deserializer throws an exception when encountering comments (because comments are not part of the official JSON standard). Setting `ReadCommentHandling` to `JsonCommentHandling.Skip` instructs the deserializer to skip over them instead, so the following can be successfully parsed:

```
{
  "Name":"Dylan" // Comment here
  /* This is another comment */
}
```

PropertyNameCaseInsensitive

By default, the deserializer is case sensitive when matching JSON property names to C# property names. This means that the following input

```
{ "name":"Dylan" }
```

would fail to populate the `Name` property in our `Person` class (the JSON property would be ignored).

Setting `PropertyNameCaseInsensitive` to `true` solves this problem by instructing the deserializer to perform case-insensitive matching (at a small performance cost):

```
var dylan = JsonSerializer.Deserialize<Person> (json,
  new JsonSerializerOptions { PropertyNameCaseInsensitive = true });
```

If the input has predictable casing, another solution is to use the `JsonPropertyName` attribute (described earlier) or the `PropertyNamingPolicy` option (described next).

> From .NET 5, the default value for the setting is `true` when writing web apps (and `false` for other kind of apps).

PropertyNamingPolicy

To better support the popular camel-case property naming convention, .NET Core 3 introduced `PropertyNamingPolicy`. It provides better performance than the just-described `PropertyNameCaseInsensitive` option and applies to both serialization and deserialization. Thus, the code

```
var dylan = new Person { Name = "Dylan" };

var json = JsonSerializer.Serialize (dylan,
  new JsonSerializerOptions
  {
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
  });
```

yields

```
{"name": "Dylan"}
```

which can be deserialized in the same way:

```
var dylan2 = JsonSerializer.Deserialize<Person> (json,
  new JsonSerializerOptions
  {
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
  });
```

From .NET 5, the `CamelCase` option is the default when writing web apps.

DictionaryKeyPolicy

With the `DictionaryKeyPolicy` option, you can force dictionary keys to serialize or deserialize with camel casing:

```
var dict = new Dictionary<string, string>
{
  { "BookName", "Nutshell" }
  { "BookVersion", "9.0" },
};

Console.WriteLine (JsonSerializer.Serialize (dict,
  new JsonSerializerOptions
  {
    WriteIndented = true,
    DictionaryKeyPolicy = JsonNamingPolicy.CamelCase
  }));
```

This outputs the following:

```
{
  "bookName": "Nutshell"
  "bookVersion": "9.0",
}
```

Encoder

The default text encoder aggressively escapes characters such that the output can appear in an HTML document without additional processing:

```
string dylan = "<b>Dylan & Friends</b>";
Console.WriteLine (JsonSerializer.Serialize (dylan));
```

Here's the output:

```
"\u003Cb\u003EDylan \u0026 Friends\u003C/b\u003E"
```

You can prevent this by changing the `Encoder`:

```
Console.WriteLine (JsonSerializer.Serialize (dylan,
  new JsonSerializerOptions {
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping
  }));
```

This yields the following output:

```
"<b>Dylan & Friends</b>"
```

`UnsafeRelaxedJsonEscaping` is a subclass of `System.Text.Encodings.Web.JavaScriptEncoder`. Should the need arise, you can implement your own subclass for complete control over the encoding process.

IgnoreNullValues

By default, `null` property values are included in the JSON output, so

```
var person = new Person { Name = null };
```

would serialize to:

```
{
  "Name": null
}
```

With `IgnoreNullValues` set to `true`, null-value properties are completely ignored:

```
Console.WriteLine (JsonSerializer.Serialize (person),
  new JsonSerializerOptions { IgnoreNullValues = true } ));
```

Here's the output:

```
{}
```

IgnoreReadOnlyProperties

By default, read-only properties are serialized (but not deserialized, because there is no set accessor). You can tell the serializer to ignore read-only properties by setting `IgnoreReadOnlyProperties` to `true`.

NumberHandling (from .NET 5)

In JSON, numbers are normally formatted without quotes. However, not all JSON writers follow this convention, and so .NET 5 introduced an option to support reading and writing numbers in quotes:

```
var options = new JsonSerializerOptions
{
  WriteIndented = true,
  NumberHandling = JsonNumberHandling.AllowReadingFromString |
                   JsonNumberHandling.WriteAsString
};

string json = JsonSerializer.Serialize (new Point { X = 2, Y = 3}, options);
Console.WriteLine (json);
var p2 = (Point) JsonSerializer.Deserialize (json, typeof (Point), options);
```

```
public class Point
{
  public int X { get; set; }
  public int Y { get; set; }
}
```

Here's the output:

```
{
  "X": "2",
  "Y": "3"
}
```

When writing web apps. `JsonNumberHandling.AllowReadingFromString` is the default, which means you can safely read numbers, whether or not they're wrapped in quotes.

The Data Contract Serializer

The data contract serializer supports a *data contract* model that helps you decouple the low-level details of the types you want to serialize from the structure of the serialized data. This provides excellent version tolerance, meaning you can deserialize data that was serialized from an earlier or later version of a type. You can even deserialize types that have been renamed or moved to a different assembly.

The data contract serializer can cope with most object graphs, although it can require more assistance than the binary serializer. It can also be used as a general-purpose tool for reading/writing XML files, if you're flexible on how the XML is structured. (If you need to store data in attributes or cope with XML elements presenting in a arbitrary order, you cannot use the data contract serializer.)

Getting Started

Here are the basic steps in using the data contract serializer:

1. Decide whether to use `DataContractSerializer` or `DataContractJsonSerializer` (in .NET Framework, there's also the `NetDataContractSerializer`).
2. Decorate the types and members you want to serialize with `[DataContract]` and `[DataMember]` attributes, respectively.
3. Instantiate the serializer and call `WriteObject` or `ReadObject`.

If you chose the `DataContractSerializer`, you will also need to register "known types" (subtypes that can also be serialized), and decide whether to preserve object references.

You may also need to take special action to ensure that collections are properly serialized.

> Types for the data contract serializer are defined in the `System.Runtime.Serialization` namespace, in an assembly of the same name.

Choosing a Serializer

There are three data contract serializers:

`DataContractSerializer`
: Loosely couples .NET types to data contract types via XML

`DataContractJsonSerializer`
: Loosely couples .NET types to data contract types via JSON

`NetDataContractSerializer`
: Tightly couples .NET types to data contract types (.NET Framework only)

The first two require that you explicitly register serializable subtypes in advance so that it can map a data contract name such as "Person" to the correct .NET type. The `NetDataContractSerializer` requires no such assistance, because it writes the full type and assembly names of the types it serializes, rather like the binary serialization engine:

```
<Person z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

Such output relies on the presence of a specific .NET type in a specific namespace and assembly in order to deserialize.

If you're saving an object graph to a "black box," you can choose any serializer, depending on what benefits are more important to you. If you're communicating through WCF, or reading/writing an XML file, you'll most likely want the `DataContractSerializer`.

We'll go into each of these topics in more detail in the following sections.

Using the Serializers

After choosing a serializer, the next step is to attach attributes to the types and members you want to serialize. At a minimum:

- Add the `[DataContract]` attribute to each type.
- Add the `[DataMember]` attribute to each member that you want to include.

Here's an example:

```
namespace SerialTest
{
  [DataContract] public class Person
  {
    [DataMember] public string Name;
    [DataMember] public int Age;
  }
}
```

These attributes are enough to make a type *implicitly* serializable through the data contract engine.

You can then *explicitly* serialize or deserialize an object instance by instantiating a serializer and calling `WriteObject` or `ReadObject`:

```
Person p = new Person { Name = "Stacey", Age = 30 };

var ds = new DataContractSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  ds.WriteObject (s, p);                          // Serialize

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) ds.ReadObject (s);                // Deserialize

Console.WriteLine (p2.Name + " " + p2.Age);       // Stacey 30
```

`DataContractSerializer`'s constructor requires the *root object* type (the type of the object you're explicitly serializing). In contrast, `NetDataContractSerializer` does not:

```
var ns = new NetDataContractSerializer();

// NetDataContractSerializer is otherwise the same to use
// as DataContractSerializer.
```

```
...
```

Both types of serializer use the XML formatter by default. With an `XmlWriter`, you can request that the output be indented for readability:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };
using (XmlWriter w = XmlWriter.Create ("person.xml", settings))
  ds.WriteObject (w, p);

System.Diagnostics.Process.Start ("person.xml");
```

Here's the result:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Stacey</Name>
</Person>
```

The XML element name `<Person>` reflects the *data contract name*, which, by default, is the .NET type name. You can override this and explicitly state a data contract name as follows:

```
[DataContract (Name="Candidate")]
public class Person { ... }
```

The XML namespace reflects the *data contract namespace*, which, by default, is *http://schemas.datacontract.org/2004/07/*, plus the .NET type namespace. You can override this in a similar fashion:

```
[DataContract (Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```

> Specifying a name and namespace decouples the contract identity from the .NET type name. It ensures that, should you later refactor and change the type's name or namespace, serialization is unaffected.

You can also override names for data members:

```
[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember (Name="FirstName")]  public string Name;
  [DataMember (Name="ClaimedAge")] public int Age;
}
```

Here's the output:

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Stacey</FirstName>
</Candidate>
```

`[DataMember]` supports both fields and properties—public and private. The field or property's data type can be any of the following:

- Any primitive type
- `DateTime`, `TimeSpan`, `Guid`, `Uri`, or an `Enum` value
- Nullable versions of the above
- `byte[]` (serializes in XML to base 64)

- Any "known" type decorated with `DataContract`
- Any `IEnumerable` type (see the section "Serializing Collections" later in this chapter)
- Any type with the `[Serializable]` attribute or implementing `ISerializable` (see the section "Extending Data Contracts" later in this chapter)
- Any type implementing `IXmlSerializable`

Specifying a binary formatter (.NET Framework only)

You can use a binary formatter with `DataContractSerializer` or `NetDataContractSerializer`. The process is the same:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

var s = new MemoryStream();
using (XmlDictionaryWriter w = XmlDictionaryWriter.CreateBinaryWriter (s))
  ds.WriteObject (w, p);

var s2 = new MemoryStream (s.ToArray());
Person p2;
using (XmlDictionaryReader r = XmlDictionaryReader.CreateBinaryReader (s2,
                               XmlDictionaryReaderQuotas.Max))
  p2 = (Person) ds.ReadObject (r);
```

The output varies between being slightly smaller than that of the XML formatter, and radically smaller if your types contain large arrays.

Serializing Subclasses

You don't need to do anything special to handle the serializing of subclasses with the `NetDataContractSerializer`. The only requirement is that subclasses have the `DataContract` attribute. The serializer will write the fully qualified names of the actual types that it serializes as follows:

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
```

A `DataContractSerializer`, however, must be informed about all subtypes that it may have to serialize or deserialize. To illustrate, suppose we subclass `Person` as follows:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
}
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

and then write a method to clone a `Person`:

```
static Person DeepClone (Person p)
{
  var ds = new DataContractSerializer (typeof (Person));
  MemoryStream stream = new MemoryStream();
  ds.WriteObject (stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject (stream);
}
```

which we call as follows:

```
Person  person  = new Person  { Name = "Stacey", Age = 30 };
```

```
Student student = new Student { Name = "Stacey", Age = 30 };
Teacher teacher = new Teacher { Name = "Stacey", Age = 30 };

Person  p2 =           DeepClone (person);     // OK
Student s2 = (Student) DeepClone (student);    // SerializationException
Teacher t2 = (Teacher) DeepClone (teacher);    // SerializationException
```

`DeepClone` works if called with a `Person` but throws an exception with a `Student` or `Teacher`, because the deserializer has no way of knowing what .NET type (or assembly) a "Student" or "Teacher" should resolve to. This also helps with security, in that it prevents the deserialization of unexpected types.

The solution is to specify all permitted or "known" subtypes. You can do this either when constructing the `DataContractSerializer`:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

or in the type itself, with the `KnownType` attribute:

```
[DataContract, KnownType (typeof (Student)), KnownType (typeof (Teacher))]
public class Person
...
```

Here's what a serialized `Student` now looks like:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        i:type="Student" >
  ...
<Person>
```

Because we specified `Person` as the root type, the root element still has that name. The actual subclass is described separately—in the `type` attribute.

> The `NetDataContractSerializer` suffers a performance hit when serializing subtypes—with either formatter. It seems that when it encounters a subtype, it has to stop and think for a while!
>
> Serialization performance matters on an application server that's handling many concurrent requests.

Object References

References to other objects are serialized, too. Consider the following classes:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}

[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Here's the result of serializing this to XML using the `DataContractSerializer`:

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
```

```
  </HomeAddress>
  <Name>...</Name>
</Person>
```

> The `DeepClone` method we wrote in the preceding section would clone `HomeAddress`, too—distinguishing it from a simple `MemberwiseClone`.

If you're using a `DataContractSerializer`, the same rules apply when subclassing `Address` as when subclassing the root type. So, if we define a `USAddress` class, for instance:

```
[DataContract]
public class USAddress : Address { }
```

and assign an instance of it to a `Person`:

```
Person p = new Person { Name = "John", Age = 30 };
p.HomeAddress = new USAddress { Street="Fawcett St", Postcode="02138" };
```

`p` could not be serialized. The solution is either to apply the `KnownType` attribute to `Address`:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

or to tell `DataContractSerializer` about `USAddress` in construction:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (USAddress) } );
```

(We don't need to tell it about `Address` because it's the declared type of the `HomeAddress` data member.)

Preserving object references

The `NetDataContractSerializer` always preserves referential equality. The `DataContractSerializer` does not, unless you specifically ask it to. (The `DataContractJsonSerializer` cannot preserve referential integrity at all.)

This means that if the same object is referenced in two different places, a `DataContractSerializer` ordinarily writes it twice. So, if we modify the preceding example so that `Person` also stores a work address:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address HomeAddress, WorkAddress;
}
```

and then serialize an instance as follows:

```
Person p = new Person { Name = "Stacey", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```

we would see the same address details twice in the XML:

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
```

```
  <Street>Odo St</Street>
</WorkAddress>
```

When this was later deserialized, `WorkAddress` and `HomeAddress` would be different objects. The advantage of this system is that it keeps the XML simple and standards-compliant. The disadvantages of this system include larger XML, loss of referential integrity, and the inability to cope with cyclical references.

You can request referential integrity by constructing a `DataContractSerializer` as follows:

```
var settings = new DataContractSerializerSettings { PreserveObjectReferences = true };
var ds = new DataContractSerializer (typeof (Person), settings);
```

You can also specify the maximum number of object references that the serializer should keep track of by assigning the `MaxItemsInObjectGraph` property of the `settings` object. The serializer throws an exception if this number is exceeded (this prevents a denial of service attack through a maliciously constructed stream).

Here's what the XML then looks like for a `Person` with the same home and work addresses:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        z:Id="1">
  <Age>30</Age>
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Stacey</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

The cost of this is in reduced interoperability (notice the proprietary namespace of the `Id` and `Ref` attributes).

Version Tolerance

You can add and remove data members without breaking forward or backward compatibility. By default, the data contract deserializers do the following:

- Skip over data for which there is no `[DataMember]` in the type.
- Won't complain if any `[DataMember]` is missing in the serialization stream.

Rather than skipping over unrecognized data, you can instruct the deserializer to store unrecognized data members in a black box, and then replay them should the type later be reserialized. This allows you to correctly round-trip data that's been serialized by a later version of your type. To activate this feature, implement `IExtensibleDataObject`. This interface really means "IBlackBoxProvider." It requires that you implement a single property, to get/set the black box:

```
[DataContract] public class Person : IExtensibleDataObject{
  [DataMember] public string Name;
  [DataMember] public int Age;

  ExtensionDataObject IExtensibleDataObject.ExtensionData { get; set; }
}
```

Required members

If a member is essential for a type, you can demand that it be present with `IsRequired`:

```
[DataMember (IsRequired=true)] public int ID;
```

If that member is not present, an exception is then thrown upon deserialization.

Member Ordering

The data contract serializers are extremely fussy about the ordering of data members. The deserializers, in fact, *skip over any members considered out of sequence*.

Members are written in the following order when serializing:

1. Base class to subclass
2. Low `Order` to high `Order` (for data members whose `Order` is set)
3. Alphabetical order (using *ordinal* string comparison)

So, in the preceding examples, `Age` comes before `Name`. In the following example, `Name` comes before `Age`:

```
[DataContract] public class Person
{
  [DataMember (Order=0)] public string Name;
  [DataMember (Order=1)] public int Age;
}
```

If `Person` has a base class, the base class's data members would all serialize first.

The main reason to specify an order is to comply with a particular XML schema. XML element order equates to data member order.

If you don't need to interoperate with anything else, the easiest approach is *not* to specify a member `Order` and rely purely on alphabetical ordering. A discrepancy will then never arise between serialization and deserialization as members are added and removed. The only time you'll come unstuck is if you move a member between a base class and a subclass.

Null and Empty Values

There are two ways to deal with a data member whose value is null or empty:

1. Explicitly write the null or empty value (the default).
2. Omit the data member from the serialization output.

In XML, an explicit null value looks like this:

```
<Person xmlns="..."
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

Writing null or empty members can waste space, particularly on a type with lots of fields or properties that are usually left empty. More importantly, you may need to follow an XML schema that expects the use of optional elements (e.g., `minOccurs="0"`) rather than `nil` values.

You can instruct the serializer not to emit data members for null/empty values as follows:

```
[DataContract] public class Person
{
  [DataMember (EmitDefaultValue=false)] public string Name;
  [DataMember (EmitDefaultValue=false)] public int Age;
}
```

`Name` is omitted if its value is `null`; `Age` is omitted if its value is `0` (the default value for the `int` type). If we were to make `Age` a nullable `int`, then it would be omitted if (and only if) its value was null.

> The data contract deserializer, in rehydrating an object, bypasses the type's constructors and field initializers. This allows you to omit data members as described without breaking fields that are assigned nondefault values through an initializer or constructor. To illustrate, suppose we set the default `Age` for a `Person` to 30 as follows:

```
[DataMember (EmitDefaultValue=false)]
public int Age = 30;
```

Now suppose that we instantiate `Person`, explicitly set its `Age` from 30 to 0, and then serialize it. The output won't include `Age`, because 0 is the default value for the `int` type. This means that in deserialization, `Age` will be ignored and the field will remain at its default value—which fortunately is 0, given that field initializers and constructors were bypassed.

Data Contracts and Collections

The data contract serializers can save and repopulate any enumerable collection. For instance, suppose we define `Person` to have a `List<>` of addresses:

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
}

[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Here's the result of serializing a `Person` with two addresses:

```
<Person ...>
  ...
  <Addresses>
    <Address>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Address>
    <Address>
      <Postcode>6152</Postcode>
      <Street>Comer St</Street>
    </Address>
  </Addresses>
  ...
</Person>
```

Notice that the serializer doesn't encode any information about the particular *type* of collection it serialized. If the `Addresses` field was instead of type `Address[]`, the output would be identical. This allows the collection type to change between serialization and deserialization without causing an error.

Sometimes, though, you need your collection to be of a more specific type than you expose. An extreme example is with interfaces:

```
[DataMember] public IList<Address> Addresses;
```

This serializes correctly (as before), but a problem arises in deserialization. There's no way the deserializer can know which concrete type to instantiate, so it chooses the simplest option—an array. The deserializer sticks to this strategy even if you initialize the field with a different concrete type:

```
[DataMember] public IList<Address> Addresses = new List<Address>();
```

(Remember that the deserializer bypasses field initializers.) The workaround is to make the data member a private field and add a public property to access it:

```
[DataMember (Name="Addresses")] List<Address> _addresses;
```

```
public IList<Address> Addresses { get { return _addresses; } }
```

In a nontrivial application, you would probably use properties in this manner anyway. The only unusual thing here is that we've marked the private field as the data member, rather than the public property.

Subclassed Collection Elements

The serializer handles subclassed collection elements transparently. You must declare the valid subtypes just as you would if they were used anywhere else:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}

public class USAddress : Address { }
```

Adding a `USAddress` to a `Person`'s address list then generates XML like this:

```
...
  <Addresses>
    <Address i:type="USAddress">
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

Customizing Collection and Element Names

If you subclass a collection class itself, you can customize the XML name used to describe each element by attaching a `CollectionDataContract` attribute:

```
[CollectionDataContract (ItemName="Residence")]
public class AddressList : Collection<Address> { }

[DataContract] public class Person
{
  ...
  [DataMember] public AddressList Addresses;
}
```

Here's the result:

```
...
  <Addresses>
    <Residence>
      <Postcode>6020</Postcode
      <Street>Odo St</Street>
    </Residence>
    ...
```

`CollectionDataContract` also lets you specify a `Namespace` and `Name`. The latter is not used when the collection is serialized as a property of another object (such as in this example), but it is when the collection is serialized as the root object.

You can also use `CollectionDataContract` to control the serialization of dictionaries:

```
[CollectionDataContract (ItemName="Entry",
                         KeyName="Kind",
                       ValueName="Number")]
public class PhoneNumberList : Dictionary <string, string> { }
```

```
[DataContract] public class Person
{
  ...
  [DataMember] public PhoneNumberList PhoneNumbers;
}
```

Here's how this formats:

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
    </Entry>
  </PhoneNumbers>
```

Extending Data Contracts

This section describes how you can extend the capabilities of the data contract serializer through serialization hooks, `[Serializable]` and `IXmlSerializable`.

Serialization and Deserialization Hooks

You can request that a custom method be executed before or after serialization, by flagging the method with one of the following attributes:

`[OnSerializing]`
: Indicates a method to be called just *before* serialization

`[OnSerialized]`
: Indicates a method to be called just *after* serialization

Similar attributes are supported for deserialization:

`[OnDeserializing]`
: Indicates a method to be called just *before* deserialization

`[OnDeserialized]`
: Indicates a method to be called just *after* deserialization

The custom method must have a single parameter of type `StreamingContext`. This parameter is required for consistency with the binary engine, and it is not used by the data contract serializer.

`[OnSerializing]` and `[OnDeserialized]` are useful in handling members that are outside the capabilities of the data contract engine, such as a collection that has an extra payload or that does not implement standard interfaces. Here's the basic approach:

```
[DataContract] public class Person
{
  public SerializationUnfriendlyType Addresses;

  [DataMember (Name="Addresses")]
  SerializationFriendlyType _serializationFriendlyAddresses;

  [OnSerializing]
```

```
  void PrepareForSerialization (StreamingContext sc)
  {
    // Copy Addresses—> _serializationFriendlyAddresses
    // ...
  }

  [OnDeserialized]
  void CompleteDeserialization (StreamingContext sc)
  {
    // Copy _serializationFriendlyAddresses—> Addresses
    // ...
  }
}
```

An `[OnSerializing]` method can also be used to conditionally serialize fields:

```
public DateTime DateOfBirth;

[DataMember] public bool Confidential;

[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;

[OnSerializing]
void PrepareForSerialization (StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

Recall that the data contract deserializers bypass field initializers and constructors. An `[OnDeserializing]` method acts as a pseudoconstructor for deserialization, and it is useful for initializing fields excluded from serialization:

```
[DataContract] public class Test
{
  bool _editable = true;

  public Test() { _editable = true; }

  [OnDeserializing]
  void Init (StreamingContext sc)
  {
    _editable = true;
  }
}
```

If it wasn't for the `Init` method, `_editable` would be false in a deserialized instance of `Test`—despite the other two attempts at making it true.

Methods decorated with these four attributes can be private. If subtypes need to participate, they can define their own methods with the same attributes, and they will get executed, too.

Interoperating with [Serializable]

The data contract serializer can also serialize types marked with the binary serialization engine's attributes and interfaces. This ability is important, since support for the binary engine has been woven into much of what was written prior to Framework 3.0—including .NET itself!

The following things flag a type as being serializable for the binary engine:

- The `[Serializable]` attribute
- Implementing `ISerializable`

Binary interoperability is useful in serializing existing types as well as new types that need to support both engines. It also provides another means of extending the capability of the data contract serializer, because the binary engine's `ISerializable` is more flexible than the data contract attributes. Unfortunately, the data contract serializer is inefficient in how it formats data added via `ISerializable`.

A type wanting the best of both worlds cannot define attributes for both engines. This creates a problem for types such as `string` and `DateTime`, which for historical reasons cannot divorce the binary engine attributes. The data contract serializer works around this by filtering out these basic types and processing them specially. For all other types marked for binary serialization, the data contract serializer applies similar rules to what the binary engine would use. This means it honors attributes such as `NonSerialized` or calls `ISerializable` if implemented. It does not *thunk* to the binary engine itself—this ensures that output is formatted in the same style as if data contract attributes were used.

Types designed to be serialized with the binary engine expect object references to be preserved. You can enable this option through the `DataContractSerializer` (or by using the `NetDataContractSerializer`).

The rules for registering known types also apply to objects and subobjects serialized through the binary interfaces.

The following example illustrates a class with a `[Serializable]` data member:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address MailingAddress;
}
[Serializable] public class Address
{
  public string Postcode, Street;
}
```

Here's the result of serializing it:

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

Had `Address` implemented `ISerializable`, the result would be less efficiently formatted:

```
  <MailingAddress>
    <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
      i:type="d3p1:string" xmlns="">str</Street>
    <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
      i:type="d3p1:string" xmlns="">pcode</Postcode>
  </MailingAddress>
```

Interoperating with IXmlSerializable

A limitation of the data contract serializer is that it gives you little control over the structure of the XML. In a WCF application this can actually be beneficial, in that it makes it easier for the infrastructure to comply with standard messaging protocols.

If you do need precise control over the XML, you can implement `IXmlSerializable` and then use `XmlReader` and `XmlWriter` to manually read and write the XML. The data contract serializer allows you to do this just on the types for which this level of control is required.

The Binary Serializer

The binary serialization engine saves and restores objects with full type and reference fidelity, and you can use it to perform such tasks as saving and restoring objects to disk. The binary serializer is highly automated and can handle complex object graphs with minimum intervention. It's not available, however, in Windows Store apps.

There are two ways to make a type support binary serialization. The first is attribute-based; the second involves implementing `ISerializable`. Adding attributes is simpler; implementing `ISerializable` is more flexible. You typically implement `ISerializable` to do the following:

- Dynamically control what gets serialized.
- Make your serializable type friendly to being subclassed by other parties.

> There are significant security issues associated with using the binary serializer. For details, see https://aka.ms/binaryformatter

Getting Started

You can make a type serializable by applying a single attribute:

```
[Serializable] public sealed class Person
{
  public string Name;
  public int Age;
}
```

The `[Serializable]` attribute instructs the serializer to include all fields in the type. This includes both private and public fields (but not properties). Every field must itself be serializable; otherwise, an exception is thrown. Primitive .NET types such as `string` and `int` support serialization (as do many other .NET types).

> The `Serializable` attribute is not inherited, so subclasses are not automatically serializable, unless also marked with this attribute.

To serialize an instance of `Person`, you instantiate `BinaryFormatter` (in `System.Runtime.Serialization.Formatters.Binary`) and call `Serialize`.

> .NET Framework also offers a `SoapFormatter` that you can use in the same way to generate SOAP-compatible XML output. It's less functional than `BinaryFormatter` and it neither supports generic types nor the filtering of extraneous data necessary for version-tolerant serialization.

The following serializes a `Person` with a `BinaryFormatter`:

```
Person p = new Person() { Name = "George", Age = 25 };

IFormatter formatter = new BinaryFormatter();

using (FileStream s = File.Create ("serialized.bin"))
  formatter.Serialize (s, p);
```

All of the data necessary to reconstruct the `Person` object is written to the file *serialized.bin*. The `Deserialize` method restores the object:

```
using (FileStream s = File.OpenRead ("serialized.bin"))
```

```
{
  Person p2 = (Person) formatter.Deserialize (s);
  Console.WriteLine (p2.Name + " " + p2.Age);     // George 25
}
```

> The deserializer bypasses all constructors and field initializers when re-creating objects. Behind the scenes, it calls `FormatterServices.GetUninitializedObject` to do this job. You can call this method yourself to implement some very grubby design patterns!

The serialized data includes full type and assembly information, so if we try to cast the result of deserialization to a matching `Person` type in a different assembly, an error would result. The deserializer fully restores object references to their original state upon deserialization. This includes collections, which are just treated as serializable objects like any other (all collection types in `System.Collections.*` are marked as serializable).

> The binary engine can handle large, complex object graphs without special assistance (other than ensuring that all participating members are serializable). One thing to be wary of is that the serializer's performance degrades in proportion to the number of references in your object graph. This can become an issue in a Remoting server that has to process many concurrent requests.

Binary Serialization Attributes

[NonSerialized]

By default, *all* fields are serialized. Fields that you don't want serialized, such as those used for temporary calculations or for storing file or window handles, you must mark explicitly with the `[NonSerialized]` attribute:

```
[Serializable] public sealed class Person
{
  public string Name;
  [NonSerialized] public int Age;
}
```

This instructs the serializer to ignore the `Age` member.

> Nonserialized members are always empty or `null` when deserialized—even if field initializers or constructors set them otherwise.

[OnDeserializing]

A method marked with the `[OnDeserializing]` attribute fires just prior to deserialization, and acts as a kind of constructor. This can be important because the binary deserializer bypasses all your normal constructors as well as field initializers.

In the following example, we define a field called `Valid` which we exclude from serialization with the `[NonSerialized]` attribute:

```
public sealed class Person
{
  public string Name;
  [NonSerialized] public bool Valid = true;

  public Person() => Valid = true;
}
```

A deserialized `Person` will never be `Valid`—despite the constructor and field initializer both setting `Valid` to `true`. We can solve this by writing a special deserialization "constructor" as follows:

```
[OnDeserializing]
void OnDeserializing (StreamingContext context) => Valid = true;
```

[OnDeserialized]

A method marked with the `[OnDeserialized]` attribute fires just *after* deserialization. This can be useful for updating calculated fields, and in conjunction with `[OnSerializing]`, which we look at next.

[OnSerializing] and [OnSerialized]

The `[OnSerializing]` and `[OnSerialized]` attributes mark methods for execution before or after serialization.

`[OnSerializing]` is useful for populating a field that's used *only* for serialization. To illustrate, suppose that you want to make the following class serializable:

```
class Foo
{
  public XDocument Xml;
}
```

The difficulty is that `XDocument` (in the `System.Xml.Linq` namespace) is not itself serializable. We can solve this by applying the `[NonSerialized]` attribute to the `Xml` field and then defining an `[OnSerializing]` method that writes the content of the `XDocument` to a string field (that we do serialize):

```
[Serializable]
class Foo
{
  [NonSerialized]
  public XDocument Xml;

  string _xmlString;  // used only for serialization

  [OnSerializing]
  void OnSerializing (StreamingContext context)
    => _xmlString = Xml.ToString();
}
```

The final step is to reconstruct the `XDocument` when deserializating. We can do this by adding an `[OnDeserialized]` method:

```
  [OnDeserialized]
  void OnDeserialized (StreamingContext context)
    => Xml = XDocument.Parse (_xmlString);
```

[OptionalField] and Versioning

Adding or removing fields doesn't break compatibility with already serialized data: the deserializer skips over data for which there's no matching field. When adding a field, you can apply the following attribute to remind yourself that it might be absent from data serialized by an older version of the software:

```
[Serializable] public sealed class Person
{
  public string Name;
  [OptionalField (VersionAdded = 2)] public DateTime DateOfBirth;
}
```

This serves as documentation, and has no effect on serialization semantics.

> If versioning robustness is important, avoid renaming fields and avoid retrospectively adding the `NonSerialized` attribute. Never change a field's type.

Binary Serialization with ISerializable

Implementing `ISerializable` gives a type complete control over its binary serialization and deserialization.

Here's the `ISerializable` interface definition:

```
public interface ISerializable
{
  void GetObjectData (SerializationInfo info, StreamingContext context);
}
```

`GetObjectData` fires upon serialization; its job is to populate the `SerializationInfo` object (a name-value dictionary) with data from all fields that you want serialized. Here's how we would write a `GetObjectData` method that serializes two fields, called `Name` and `DateOfBirth`:

```
public virtual void GetObjectData (SerializationInfo info,
                                   StreamingContext context)
{
  info.AddValue ("Name", Name);
  info.AddValue ("DateOfBirth", DateOfBirth);
}
```

In this example, we've chosen to name each item according to its corresponding field. This is not required; you can use any name, but you must use the same name upon deserialization. The values themselves can be of any serializable type; the serialization will continue recursively as necessary. It's legal to store null values in the dictionary.

> It's a good idea to make the `GetObjectData` method `virtual`—unless your class is `sealed`. This allows subclasses to extend serialization without having to reimplement the interface.

`SerializationInfo` also contains properties that you can use to control the type and assembly into which the instance should deserialize.

In addition to implementing `ISerializable`, a type controlling its own serialization needs to provide a deserialization constructor that takes the same two parameters as `GetObjectData`. The constructor can be declared with any accessibility and the runtime will still find it. Typically, though, you would declare it `protected` so that subclasses can call it.

In the following example, we define `Player` and `Team` classes, following the principles of immutability (with everything read-only). But because the immutable collections are not serializable, we need to take control over the serialization process by implementing `ISerializable`:

```
[Serializable] public class Player
{
  public readonly string Name;
  public Player (string name) => Name = name;
}

[Serializable] public class Team : ISerializable
{
  public readonly string Name;
  public readonly ImmutableList<Player> Players;   // Not serializable!

  public Team (string name, params Player[] players)
  {
    Name = name;
    Players = players.ToImmutableList();
  }

  // Serialize the object:
  public virtual void GetObjectData (SerializationInfo si, StreamingContext sc)
```

```
  {
    si.AddValue ("Name", Name);
    // Convert Players to an ordinary serializable array:
    si.AddValue ("PlayerData", Players.ToArray());
  }

  // Deserialize the object:
  protected Team (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");

    // Deserialize Players to an array to match our serialization:
    Player[] p = (Player[]) si.GetValue ("PlayerData", typeof (Player[]));

    // Construct a new immutable List using this array:
    Players = p.ToImmutableList();
  }
}
```

(You could also solve this problem by using the `[OnSerializing]` and `[OnDeserialized]` attributes that we discussed earlier.)

For commonly used types, the `SerializationInfo` class has typed "Get" methods such as `GetString`, in order to make writing deserialization constructors easier. If you specify a name for which no data exists, an exception is thrown. This happens most often when there's a version mismatch between the code doing the serialization and deserialization. You've added an extra field, for instance, and then forgotten about the implications of deserializing an old instance. To work around this problem, you can do either of the following:

- Add exception handling around code that retrieves a data member added in a later version.
- Implement your own version numbering system; for example:

```
public string MyNewField;

public virtual void GetObjectData (SerializationInfo si, StreamingContext sc)
{
  si.AddValue ("_version", 2);
  si.AddValue ("MyNewField", MyNewField);
  ...
}

protected Team (SerializationInfo si, StreamingContext sc)
{
  int version = si.GetInt32 ("_version");
  if (version >= 2) MyNewField = si.GetString ("MyNewField");
  ...
}
```

Subclassing Serializable Classes

In the preceding examples, we `sealed` the classes that relied on attributes for serialization. To see why, consider the following class hierarchy:

```
[Serializable] public class Person
{
  public string Name;
  public int Age;
}

[Serializable] public sealed class Student : Person
{
```

```
  public string Course;
}
```

In this example, both `Person` and `Student` are serializable, and both classes use the default runtime serialization behavior because neither class implements `ISerializable`.

Now imagine that the developer of `Person` decides for some reason to implement `ISerializable` and provide a deserialization constructor to control `Person` serialization. The new version of `Person` might look like this:

```
[Serializable] public class Person : ISerializable
{
  public string Name;
  public int Age;

  public virtual void GetObjectData (SerializationInfo si, StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    si.AddValue ("Age", Age);
  }

  protected Person (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");
    Age = si.GetInt32 ("Age");
  }

  public Person() {}
}
```

Although this works for instances of `Person`, this change breaks serialization of `Student` instances. Serializing a `Student` instance would appear to succeed, but the `Course` field in the `Student` type isn't saved to the stream because the implementation of `ISerializable.GetObjectData` on `Person` has no knowledge of the members of the `Student`-derived type. Additionally, deserialization of `Student` instances throws an exception because the runtime is looking (unsuccessfully) for a deserialization constructor on `Student`.

The solution to this problem is to implement `ISerializable` from the outset for serializable classes that are public and nonsealed. (With `internal` classes, it's not so important because you can easily modify the subclasses later if required.)

If we started out by writing `Person`, as in the preceding example, `Student` would then be written, as follows:

```
[Serializable]
public class Student : Person
{
  public string Course;

  public override void GetObjectData (SerializationInfo si, StreamingContext sc)
  {
    base.GetObjectData (si, sc);
    si.AddValue ("Course", Course);
  }

  protected Student (SerializationInfo si, StreamingContext sc)
    : base (si, sc)
  {
    Course = si.GetString ("Course");
  }

  public Student() {}
}
```

Chapter 27

The Roslyn Compiler

The C# compiler is itself written in C# and available as a set of modular libraries known as "Roslyn." By referencing these libraries, you can utilize the compiler's functionality in many ways besides compiling source code to an assembly. For example, you can write static code analysis and refactoring tools, editors with syntax highlighting and code completion, and Visual Studio plug-ins that understand C# code.

You can download the Roslyn libraries from NuGet, and there are packages for both C# and Visual Basic. Because both languages share some architecture, there are common dependencies. The NuGet package ID for the C# compiler libraries is Microsoft.CodeAnalysis.CSharp.

Roslyn's GitHub site also includes documentation, examples, and walkthroughs that demonstrate code analysis and refactoring.

Roslyn Architecture

The Roslyn architecture separates compilation into three phases:

1. Parsing code into syntax trees (the *syntactic* layer)
2. Binding identifiers to symbols (the *semantic* layer)
3. Emitting Intermediate Language (IL)

In the first phase, a *parser* reads C# code and outputs *syntax trees*. A syntax tree is a Document Object Model (DOM) that describes source code in tree structure.

The second phase is the one in which C#'s *static binding* takes place. Assembly references are read, and the compiler determines, for instance, that "Console" refers to `System.Console` in *System.Console.dll*. Overload resolution and type inference are a part of this, too.

The third phase produces the output assembly. If you plan to use Roslyn for code analysis or refactoring, you won't use this functionality.

Visual Studio's editor uses the output of the syntactic layer to color keywords, strings, comments and disabled code (in blue, red, green, and gray, respectively), whereas it uses the output of the semantic layer to color resolved type names (in turquoise).

Workspaces

In this chapter, we describe the compiler and the features it exposes. It's worth keeping in mind that there are additional "layers" above the compiler, including *workspaces* and *features*.

The workspaces layer is shipped in the *Microsoft.CodeAnalysis.CSharp.Workspaces* NuGet package and provides APIs to work with solutions, projects, and documents.

The features layer is shipped in *Microsoft.CodeAnalysis.CSharp.Features* and includes numerous APIs for code analysis and refactoring.

Scripting

With the *Microsoft.CodeAnalysis.CSharp.Scripting* NuGet package, you can write code such as the following:

```
int result = (int) await CSharpScript.EvaluateAsync ("1 + 2");
```

Behind the scenes, the scripting API compiles "1 + 2" into a program that it then executes, so it's less efficient than the solution that we describe in Chapter 19 (see "Interoperating with Dynamic Languages"). There are more examples on how to use the Roslyn scripting API at *https://github.com/dotnet/roslyn/wiki/Scripting-API-Samples*.

Syntax Trees

A syntax tree is a DOM for source code. The syntax tree API is completely separate from the `System.Linq.Expressions` API we discussed in "Expression Trees" in Chapter 8, although the two have conceptual similarities. Both APIs can represent C# expressions in a DOM; however, a Roslyn syntax tree has the following unique features:

- It can represent the entire C# language, not just expressions.
- It can include comments, whitespace, and other "trivia" and can round-trip with full fidelity back to the original source code.
- It comes with a `ParseText` method that parses source code into a syntax tree.

Conversely, the `System.Linq.Expressions` API has the following unique features:

- It's built into .NET, and the C# compiler itself is programmed to emit `System.Linq.Expression` types when it encounters a lambda expression with an assignment conversion to `Expression<T>`.
- It has a fast and lightweight `Compile` method that emits a delegate. In contrast, the semantic layer that compiles Roslyn syntax trees offers only the heavyweight option of compiling a complete program into an assembly.

Something that both APIs have in common is that syntax trees are immutable, so none of its elements can be altered after it's created. This means that applications such as Visual Studio and LINQPad must create a new syntax tree each time you press a key in the editor in order to update syntax highlighting and autocompletion services. This is less expensive than it sounds because the new syntax tree is able to reuse most of the elements of the old (see "Transforming a Syntax Tree"). And knowing that an object cannot change makes the API simpler to work with. It also allows for easier and faster parallelization because multithreaded code can safely access all parts of a syntax tree without locks.

SyntaxTree Structure

A `SyntaxTree` comprises three main elements:

Nodes
: (Abstract `SyntaxNode` class) Represents C# constructs such as expressions, statements, method declarations. Nodes always have at least one child, so a node can never be a leaf in the tree. Nodes can have both nodes and tokens as children.

Tokens
: (`SyntaxToken` struct) Represents the identifiers, keywords, operators, and punctuation that make up your source code. The only kind of children that tokens can have is optional leading and trailing trivia. A token's parent is always a node.

Trivia
: (`SyntaxTrivia` struct) Trivia is for whitespace, comments, preprocessor directives, and code that's inactive due to conditional compilation. Trivia is always associated with the token that's immediately to its left or right, and is exposed via that token's `TrailingTrivia` and `LeadingTrivia` properties, respectively.

Figure 27-1 shows the structure of the following code, with nodes in black, tokens in gray, and trivia in white:

```
Console.WriteLine ("Hello");
```

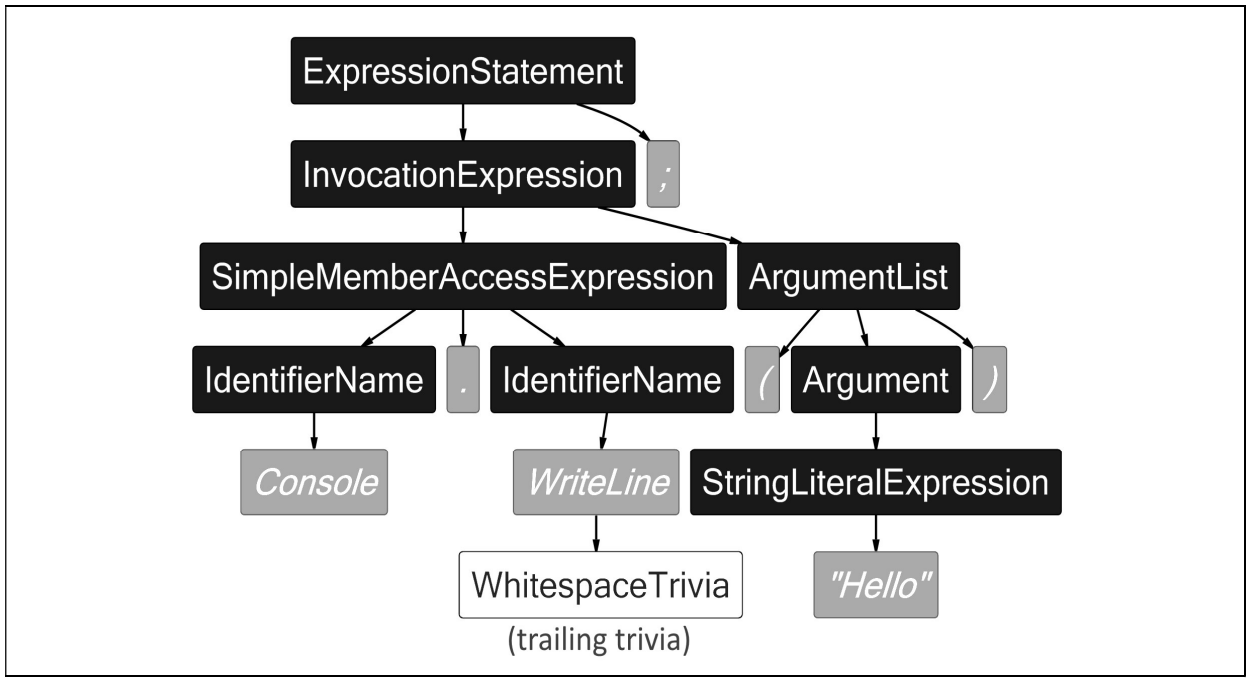

Figure 27-1. Syntax trees

`SyntaxNode` is abstract and has a C#-specific subclass for each kind of syntactic element, such as `VariableDeclarationSyntax` or `TryStatementSyntax`.

`SyntaxToken`/`SyntaxTrivia` are structs, and so a single type represents every kind of token/trivia. To distinguish different kinds of token or trivia, you must use the `RawKind` property or `Kind` extension method (which we explain in the following section).

> The best way to explore a syntax tree is with a visualizer. Visual Studio has a downloadable visualizer for use with its debugger, and LINQPad has one built in. LINQPad displays the visualizer automatically for the code in the text editor when you click the Tree button in the output window. You can also ask LINQPad to display a visualizer for a syntax tree that you've created programmatically by calling `DumpSyntaxTree` on the tree (or `DumpSyntaxNode` on a node).

Understanding Node Types

The subclasses of `SyntaxNode` have been designed to reflect the result of syntactical parsing, and are blind to semantic type/symbol information obtained from binding that occurs later. For example, consider the result of parsing the following code:

```
using System;

class Foo : SomeBaseClass
{
  void Test() { Console.WriteLine(); }
}
```

You might expect `Console.WriteLine` to be represented by a class called `MethodCallExpressionSyntax`, but no such class exists. Instead, it's represented by an `InvocationExpressionSyntax`, under which there's a `SimpleMemberAccessExpression`. This is because the parser is ignorant of types, so it cannot know that `Console` is a type, and `WriteLine` is a method. There are many other possibilities: `Console` could be a property of `SomeBaseClass`, or `WriteLine` could be an event, field, or property of a delegate type. All we can know from the syntax is that we're performing a member access (*identifier.identifier*), followed by some kind of *invocation* with zero arguments.

Common properties and methods

Nodes, tokens, and trivia have a number of important common properties and methods:

`SyntaxTree` property
: Returns the syntax tree to which the object belongs.

`Span` property
: Returns the object's position in source code (see "Finding a child by its offset").

`Kind` extension method
: Returns a `SyntaxKind` enum that classifies the node, token or trivia into one of several hundred values (e.g., `IntKeyword`, `CommaToken`, and `WhitespaceTrivia`). The same `SyntaxKind` enum covers nodes, tokens, and trivia.

`ToString` method
: Returns the text (source code) for the node, token, or trivia. For tokens, the `Text` property is equivalent.

`GetDiagnostics` method
: Returns errors or warnings generated during parsing.

`IsEquivalentTo` method
: Returns true if the object is identical to another node, token or trivia instance. Whitespace differences are significant (to ignore whitespace, call `NormalizeWhitespace` before comparing).

> Nodes and tokens also have a `FullSpan` property and `ToFullString` method. These take into account trivia, whereas `Span` and `ToString` do not.

The `Kind` extension method is a shortcut for casting the `RawKind` property, which is of type `int`, to `Microsoft.CodeAnalysis.CSharp.SyntaxKind`. The reason for not simply having a `Kind` property of type `SyntaxKind` is that the token and trivia types are also used in Visual Basic syntax trees, which has a different enum type for `SyntaxKind`.

Obtaining a Syntax Tree

The static `ParseText` method on `CSharpSyntaxTree` parses C# code into a `SyntaxTree`:

```
SyntaxTree tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Hello"");
}");

Console.WriteLine (tree.ToString());

tree.DumpSyntaxTree();    // Displays Syntax Tree Visualizer in LINQPad
```

To run this in a Visual Studio project, install the *Microsoft.CodeAnalysis.CSharp* NuGet package, and import the following namespaces:

```
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
```

You can optionally pass in a `CSharpParseOptions` object to specify a C# language version, preprocessor symbols, and a `DocumentationMode` to indicate whether XML comments should be parsed (see "Structured Trivia"). There's also an option to specify a `SourceCodeKind`. Choosing `Script` instructs the parser to accept a single expression or statement(s) instead of requiring an entire program (supported in Roslyn version 2 and later).

Another way to obtain a syntax tree is to call `CSharpSyntaxTree.Create`, passing in an object graph of nodes and tokens. We describe how to create these objects in "Transforming a Syntax tree."

After parsing a tree, you can obtain errors and warnings by calling `GetDiagnostics`. (You can also call this method on a specific node or token.)

> If the parse resulted in unexpected errors, the tree's structure may not be as you expect. For this reason, it's worth calling `GetDiagnostics` before proceeding further.

A nice feature is that a tree with errors will round-trip back to the original text (with the same errors). In such cases, the parser does its best to provide a syntax tree that's useful to the semantic layer, creating "phantom nodes" if necessary. This allows tools such as code completion to work with incomplete code. (You can determine whether a node is phantom by checking the `IsMissing` property.)

Calling `GetDiagnostics` on the syntax tree we created in the last section indicates no errors, despite having called `Console.WriteLine` without importing the `System` namespace. This is a good example of syntactic versus semantic parsing: our program is syntactically correct, and our error will not manifest until we create a compilation, add assembly references, and query the *semantic model*, where binding takes place.

Traversing and Searching a Tree

A `SyntaxTree` acts as a wrapper for the tree structure. It has a reference to a single root node, which you obtain by calling `GetRoot`:

```
var tree = CSharpSyntaxTree.ParseText (@"class Test
{
  static void Main() => Console.WriteLine (""Hello"");
}");

SyntaxNode root = tree.GetRoot();
```

The root node of a C# program is a `CompilationUnitSyntax`:

```
Console.WriteLine (root.GetType().Name);   // CompilationUnitSyntax
```

Traversing children

`SyntaxNode` exposes LINQ-friendly methods to traverse its child nodes and tokens. Here are the simplest:

```
IEnumerable<SyntaxNode> ChildNodes()
IEnumerable<SyntaxToken> ChildTokens()
```

Following on from our previous example, our root node has a single child node of type `ClassDeclarationSyntax`:

```
var cds = (ClassDeclarationSyntax) root.ChildNodes().Single();
```

We can enumerate the members of `cds` via either its `ChildNodes` method or the `Members` property of `ClassDeclarationSyntax`

```
foreach (MemberDeclarationSyntax member in cds.Members)
  Console.WriteLine (member.ToString());
```

with the following result:

```
static void Main() => Console.WriteLine (""Hello"");
```

There are also `Descendant*` methods which descend recursively into children. We can enumerate the tokens that make up our program, as follows:

```
foreach (var token in root.DescendantTokens())
  Console.WriteLine ($"{token.Kind(),-30} {token.Text}");
```

Here's the result:

```
ClassKeyword                   class
IdentifierToken                Test
OpenBraceToken                 {
StaticKeyword                  static
VoidKeyword                    void
IdentifierToken                Main
OpenParenToken                 (
CloseParenToken                )
EqualsGreaterThanToken         =>
IdentifierToken                Console
DotToken                       .
IdentifierToken                WriteLine
OpenParenToken                 (
StringLiteralToken             "Hello"
CloseParenToken                )
SemicolonToken                 ;
CloseBraceToken                }
EndOfFileToken
```

Notice that there's no whitespace in the result. Replacing `token.Text` with `token.ToFullString()` would give us whitespace (and any other trivia).

The following uses the `DescendantNodes` method to locate the syntax node for our method declaration:

```
var ourMethod = root.DescendantNodes()
                    .First (m => m.Kind() == SyntaxKind.MethodDeclaration);
```

Or, alternatively:

```
var ourMethod = root.DescendantNodes()
                    .OfType<MethodDeclarationSyntax>()
                    .Single();
```

With the latter example, `ourMethod` is of type `MethodDeclarationSyntax` which exposes useful properties specific to method declarations. For instance, if our example contained more than one method definition and we wanted to find just the method whose name is "Main", we could do this:

```
var mainMethod = root.DescendantNodes()
                     .OfType<MethodDeclarationSyntax>()
                     .Single (m => m.Identifier.Text == "Main");
```

`Identifier` is a property on `MethodDeclarationSyntax` that returns the token corresponding to the method's identifier (i.e., its name). We could get the same result with more effort, as follows:

```
root.DescendantNodes().First (m =>
  m.Kind() == SyntaxKind.MethodDeclaration &&
```

```
m.ChildTokens().Any (t =>
  t.Kind() == SyntaxKind.IdentifierToken && t.Text == "Main"));
```

`SyntaxNode` also has `GetFirstToken` and `GetLastToken` methods which are equivalent to calling `DescendantTokens().First()` and `DescendantTokens().Last()`.

> `GetLastToken()` is faster than `DescendantTokens().Last()` because it returns a direct link rather than enumerating through all descendants.

As nodes can contain both child nodes and tokens whose relative order is significant, there are also methods to enumerate both together:

```
ChildSyntaxList ChildNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokens()
IEnumerable<SyntaxNodeOrToken> DescendantNodesAndTokensAndSelf()
```

(`ChildSyntaxList` implements `IEnumerable<SyntaxNodeOrToken>` while also exposing a `Count` property and an indexer to access an element by position.)

You can traverse trivia directly from a node with the `GetLeadingTrivia`, `GetTrailingTrivia` and `DescendantTrivia` methods. More commonly, though, you'd access trivia through the token to which it's attached via the token's `LeadingTrivia` and `TrailingTrivia` properties. Or, to convert to text, you'd use the `ToFullString` method which includes trivia in the result.

Traversing parents

Nodes and tokens have a `Parent` property of type `SyntaxNode`.

For `SyntaxTrivia`, the "parent" is its token, accessible via the `Token` property.

Nodes also have methods that ascend back up the tree; these are prefixed with `Ancestor`.

Finding a child by its offset

All nodes, tokens and trivia have a `Span` property of type `TextSpan` to indicate starting and ending offsets in the source code. Nodes and tokens also have a `FullSpan` property which includes leading and trailing trivia (whereas `Span` does not). A node's `Span` does, however, include child nodes and tokens.

Working with TextSpan

The `TextSpan` struct has `Start`, `Length`, and `End` integer properties, which indicate character offsets in the source code. It also has methods such as `Overlap`, `OverlapsWith`, `Intersection`, and `IntersectsWith`. The difference between overlapping and intersecting is a matter of one character: two spans *overlap* if one starts before the other ends (`<`), whereas they *intersect* if they merely touch (`<=`).

The `SyntaxTree` class exposes a `GetLineSpan` method which converts a `TextSpan` into a line and character offset. This method ignores the effects of any `#line` directives present in the source code. There's also a `GetMappedLineSpan` method that takes these directives into account.

You can find a descendant object by position by calling the `FindNode`, `FindToken`, and `FindTrivia` methods on `SyntaxNode`. These methods return the descendant object with the smallest span that fully contains the span that you specify. There's also a `ChildThatContainsPosition` method which searches both descendant nodes and tokens.

Should a search result in two nodes with an identical span (typically a child and grandchild), the `FindNode` method will return the outer (parent) node. You can change this behavior by passing `true` to the optional argument `getInnermostNodeForTie`.

The `Find*` methods also have an optional `findInsideTrivia bool` parameter. If true, this also searches for nodes or tokens within *structured trivia* (see "Trivia").

CSharpSyntaxWalker

Another way to traverse a tree is by subclassing `CSharpSyntaxWalker`, overriding one or more of its hundreds of virtual methods. This following class counts the number of `if` statements:

```
class IfCounter : CSharpSyntaxWalker
{
  public int IfCount { get; private set; }

  public override void VisitIfStatement (IfStatementSyntax node)
  {
    IfCount++;
    // Call the base method if you want to descend into children.
    base.VisitIfStatement (node);
  }
}
```

Here's how to invoke it:

```
var ifCounter = new IfCounter ();
ifCounter.Visit (root);
Console.WriteLine ($"I found {ifCounter.IfCount} if statements");
```

The result is equivalent to the following:

```
root.DescendantNodes().OfType<IfStatementSyntax>().Count()
```

Writing a syntax walker can be easier than using the `Descendant*` methods in more complex cases when you need to override multiple methods (in part, because C# has no F#-like pattern matching ability).

By default, `CSharpSyntaxWalker` visits just nodes. To visit tokens or trivia, you must call the base constructor with a `SyntaxWalkerDepth`, indicating the desired depth (node→token→trivia). Then, you can override `VisitToken` and `VisitTrivia`:

```
class WhiteWalker : CSharpSyntaxWalker   // Counts space characters
{
  public int SpaceCount { get; private set; }

  public WhiteWalker() : base (SyntaxWalkerDepth.Trivia) { }

  public override void VisitTrivia (SyntaxTrivia trivia)
  {
    SpaceCount += trivia.ToString().Count (char.IsWhiteSpace);
    base.VisitTrivia (trivia);
  }
}
```

If you remove `WhiteWalker`'s call to the base constructor, `VisitTrivia` will not fire.

Trivia

Trivia is for code that, after parsing, the compiler can almost entirely ignore in terms of producing an output assembly. This comprises whitespace, comments, XML documentation, preprocessor directives, and code that's inactive by virtue of conditional compilation.

The mandatory whitespace in your code is also considered trivia. Although essential for parsing, it's not needed once the syntax tree has been produced (at least by the compiler). Trivia is still important for round-tripping back to the original source code.

Trivia belongs to the token to which it's adjacent. By convention, the parser puts whitespace and comments that follow a token, up to the end of the line, into the token's trailing trivia. Anything after that, it treats as leading trivia for the next token. (There are exceptions for the very start/end of the file.) If you're creating tokens programmatically (see "Transforming a Syntax tree"), you can put the whitespace in either place (or not at all, if you're not going to convert back to source code):

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
    static /*comment*/ void Main() {}
}");

SyntaxNode root = tree.GetRoot();

// Find the static keyword token:
var method = root.DescendantTokens().Single (t =>
  t.Kind() == SyntaxKind.StaticKeyword);

// Print out the trivia around the static keyword token:
foreach (SyntaxTrivia t in method.LeadingTrivia)
  Console.WriteLine (new { Kind = "Leading " + t.Kind(), t.Span.Length });

foreach (SyntaxTrivia t in method.TrailingTrivia)
  Console.WriteLine (new { Kind = "Trailing " + t.Kind(), t.Span.Length });
```

Here's the output:

```
{ Kind = Leading WhitespaceTrivia, Length = 1 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
{ Kind = Trailing MultiLineCommentTrivia, Length = 11 }
{ Kind = Trailing WhitespaceTrivia, Length = 1 }
```

Preprocessor directives

It might seem odd that preprocessor directives are considered trivia given that some directives (in particular, conditional compilation directives) have a nontrivial effect on the output.

The reason is that preprocessor directives are processed semantically by the parser itself; that is, it's the parser's job to do the preprocessing. After which, there's nothing left that the compiler need explicitly consider (except for `#pragma`). To illustrate, let's examine how the parser handles conditional compilation directives:

```
#define FOO

#if FOO
    Console.WriteLine ("FOO is defined");
#else
    Console.WriteLine ("FOO is not defined");
#endif
```

Upon reading the `#if FOO` directive, the parser knows that `FOO` is defined, and so the line that follows is parsed normally (as nodes and tokens), whereas the line of code following the `#else` directive is parsed into `DisabledTextTrivia`.

> When calling `CSharpSyntaxTree.Parse`, you can supply additional preprocessor symbols by constructing and passing in a `CSharpParseOptions` instance.

Hence, with conditional compilation, it is precisely the text that can be ignored that ends up in trivia (i.e., the inactive code and the preprocessor directives themselves).

The `#line` directive is handled similarly, in that the parser reads and interprets the directive. The information that it harvests is used when you call `GetMappedLineSpan` on the syntax tree.

The `#region` directive is semantically empty: the only role of the parser is to check that `#region` directives are matched with `#endregion` directives. The `#error` and `#warning` directives are also processed by the parser, which generates errors and warnings that you can see by calling `GetDiagnostics` on the tree or node.

It can be still useful to examine the content of preprocessor directives for purposes other than producing the output assembly (syntax highlighting, for instance). This is made easier through *structured trivia*.

Structured trivia

There are two kinds of trivia:

Unstructured trivia
: Comments, whitespace, and code that's inactive due to conditional compilation

Structured trivia
: Preprocessor directives and XML documentation

Unstructured trivia is treated purely as text, whereas structured trivia also has its content parsed into a miniature syntax tree.

The `HasStructure` property on `SyntaxTrivia` indicates whether structured trivia is present, and the `GetStructure` method returns the root node for the miniature syntax tree:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");

// In LINQPad:
tree.DumpSyntaxTree();  // LINQPad displays structured trivia in Visualizer

SyntaxNode root = tree.GetRoot();

var trivia = root.DescendantTrivia().First();
Console.WriteLine (trivia.HasStructure);           // True
Console.WriteLine (trivia.GetStructure().Kind());  // DefineDirectiveTrivia
```

In the case of preprocessor directives, you can navigate directly to the structured trivia by calling `GetFirstDirective` on a `SyntaxNode`. There's also a `ContainsDirectives` property to indicate whether preprocessor trivia is present:

```
var tree = CSharpSyntaxTree.ParseText (@"#define FOO");
SyntaxNode root = tree.GetRoot();

Console.WriteLine (root.ContainsDirectives);      // True

// directive is the root node of the structured trivia:
var directive = root.GetFirstDirective();
Console.WriteLine (directive.Kind());             // DefineDirectiveTrivia
Console.WriteLine (directive.ToString());         // #define FOO

// If there were more directives, we could get to them as follows:
Console.WriteLine (directive.GetNextDirective());    // (null)
```

After we have a trivia node, we can cast it to a specific type and query its properties, just as we would with any other node:

```
var hashDefine = (DefineDirectiveTriviaSyntax) root.GetFirstDirective();
Console.WriteLine (hashDefine.Name.Text);     // FOO
```

> All nodes, tokens and trivia have the `IsPartOfStructuredTrivia` property to indicate whether the object in question is part of a structured trivia tree (i.e., descends from a trivia object).

Transforming a Syntax Tree

You can "modify" nodes, tokens, and trivia via a set of methods with the following prefixes (most of which are extension methods):

```
Add*
Insert*
Remove*
Replace*
With*
Without*
```

Because syntax trees are immutable, all of these methods return a new object with the desired modifications, leaving the original untouched.

Handling changes to the source code

If you're writing a C# editor, for instance, you'll need to update a syntax tree based on changes to the source code. The `SyntaxTree` class has a `WithChangedText` method that does exactly this: it partially reparses the source code based on modifications that you describe with a `SourceText` instance (in `Microsoft.CodeAnalysis.Text`).

To create a `SourceText`, use its static `From` method, giving it the complete source code. You then can use this to create a syntax tree:

```
SourceText sourceText = SourceText.From ("class Program {}");
var tree = CSharpSyntaxTree.ParseText (sourceText);
```

Alternatively, you can obtain the `SourceText` for an existing tree by calling `GetText`.

You now can "update" `sourceText` by calling `Replace` or `WithChanges`. For example, we could replace the first five characters (`class`) with `struct`, as follows:

```
var newSource = sourceText.Replace (0, 5, "struct");
```

Finally, we can call `WithChangedText` on the tree to update it:

```
var newTree = tree.WithChangedText (newSource);
Console.WriteLine (newTree.ToString());         // struct Program {}
```

Creating new nodes, tokens and trivia with SyntaxFactory

The static methods on `SyntaxFactory` programmatically create nodes, tokens, and trivia, which you can use to "transform" existing syntax trees or to create new trees from scratch.

The most difficult part of doing this is establishing exactly what kind of nodes and tokens to create. The solution is to first parse a sample of the code you want, examining the result in a syntax visualizer. For instance, suppose that we want to create a syntax node for the following:

```
using System.Text;
```

We can visualize the syntax tree for this in LINQPad, as follows:

```
CSharpSyntaxTree.ParseText ("using System.Text;").DumpSyntaxTree();
```

(We can parse `using System.Text;` without error because it's valid as a complete program, albeit a functionally empty one. For most other code snippets, you'll need to wrap the snippet in a method and/or type definition so that it will parse.)

The result has the following structure, of which we are interested in the second node—`UsingDirective` and its descendants:

```
Kind                              Token Text
================================  ==========
CompilationUnit (node)
  UsingDirective (node)
    UsingKeyword (token)          using
```

```
      WhitespaceTrivia (trailing)
    QualifiedName (node)
      IdentifierName (node)
        IdentifierToken (token)    System
      DotToken (token)             .
      IdentifierName (node)
        IdentifierToken (token)    Text
    SemiColonToken (token)         ;
  EndOfFileToken (token)
```

Starting from the inside, we have two `IdentifierName` nodes, whose parent is a `QualifiedName`. We can create that, as follows:

```
QualifiedNameSyntax qualifiedName = SyntaxFactory.QualifiedName (
  SyntaxFactory.IdentifierName ("System"),
  SyntaxFactory.IdentifierName ("Text"));
```

We used the overload of `QualifiedName` that accepts two identifiers. This overload inserts the dot token for us automatically.

We now need to wrap this in a `UsingDirective`:

```
UsingDirectiveSyntax usingDirective =
  SyntaxFactory.UsingDirective (qualifiedName);
```

Because we didn't specify tokens for the `using` keyword or the trailing semicolon, tokens for each were automatically created and added. However, the automatically created tokens don't include whitespace. This wouldn't prevent compilation, but converting the tree to a string would result in syntactically incorrect code:

```
Console.WriteLine (usingDirective.ToFullString());  // usingSystem.Text;
```

We can fix this by calling `NormalizeWhitespace` on the node (or one of its ancestors); doing so automatically adds whitespace trivia (for both syntactic correctness and readability). Or for more control, we could add whitespace explicitly:

```
usingDirective = usingDirective.WithUsingKeyword (
  usingDirective.UsingKeyword.WithTrailingTrivia (
    SyntaxFactory.Whitespace (" ")));

Console.WriteLine (usingDirective.ToFullString());  // using System.Text;
```

For brevity, we "harvested" the node's existing `UsingKeyword` to which we added trailing trivia. We could have created an equivalent token with more effort by calling `SyntaxFactory.Token(SyntaxKind.UsingKeyword)`.

The final step is to add our `UsingDirective` node to an existing or new syntax tree (or more precisely, the root node of a tree). To do the former, we cast the existing tree's root to a `CompilationUnitSyntax` and call the `AddUsings` method. We then can create a new tree from the transformed compilation unit:

```
var existingTree = CSharpSyntaxTree.ParseText ("class Program {}");
var existingUnit = (CompilationUnitSyntax) existingTree.GetRoot();

var unitWithUsing = existingUnit.AddUsings (usingDirective);

var treeWithUsing = CSharpSyntaxTree.Create (
  unitWithUsing.NormalizeWhitespace());
```

> Remember that all parts of a syntax tree are immutable. Calling `AddUsings` returns a new node, leaving the original untouched. Ignoring the return value is an easy mistake to make!

We called `NormalizeWhitespace` on our compilation unit so that calling `ToString` on the tree will yield syntactically correct and readable code. Alternatively, we could have added explicit new-line trivia to `usingDirective`, as follows:

```
.WithTrailingTrivia (SyntaxFactory.EndOfLine("\r\n\r\n"))
```

Creating a compilation unit and syntax tree from scratch is a similar process. The easiest approach is to start with an empty compilation unit and call `AddUsings` on the unit as we did before:

```
var unit = SyntaxFactory.CompilationUnit().AddUsings (usingDirective);
```

We can add type definitions to our compilation unit by creating them in a similar fashion, and then calling `AddMembers`:

```
// Create a simple empty class definition:
unit = unit.AddMembers (SyntaxFactory.ClassDeclaration ("Program"));
```

The final step is to create the tree:

```
var tree = CSharpSyntaxTree.Create (unit.NormalizeWhitespace());
Console.WriteLine (tree.ToString());

// Output:
using System.Text;

class Program
{
}
```

CSharpSyntaxRewriter

For more complex syntax tree transformations, you can subclass `CSharpSyntaxRewriter`.

`CSharpSyntaxRewriter` is similar to the `CSharpSyntaxWalker` class that we looked at previously (see "CSharpSyntaxWalker") except that each `Visit*` method accepts and returns a syntax node. By returning something other than was passed in, you can "rewrite" the syntax tree.

For instance, the following rewriter changes method declaration names to uppercase:

```
class MyRewriter : CSharpSyntaxRewriter
{
  public override SyntaxNode VisitMethodDeclaration
    (MethodDeclarationSyntax node)
  {
    // "Replace" the method's identifier with an uppercase version:
    return node.WithIdentifier (
      SyntaxFactory.Identifier (
        node.Identifier.LeadingTrivia,           // Preserve old trivia
        node.Identifier.Text.ToUpperInvariant(),
        node.Identifier.TrailingTrivia));        // Preserve old trivia
  }
}
```

Here's how to use it:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() { Test(); }
  static void Test() {         }
}");

var rewriter = new MyRewriter();
var newRoot = rewriter.Visit (tree.GetRoot());
Console.WriteLine (newRoot.ToFullString());

// Output:
class Program
{
  static void MAIN() { Test(); }
  static void TEST() {         }
```

```
}
```

Notice that our call to `Test()` in the main method did not get renamed, because we visited just member *declarations* and ignored *invocations*. To reliably rename invocations, however, we must be able to determine whether calls to `Main()` or `Test()` refer to the `Program` type, and not some other type. To do this, a syntax tree is not enough on its own; we also need a *semantic model*.

Compilations and Semantic Models

A compilation comprises syntax trees, references, and compilation options. It serves two purposes:

- Allows compilation to a library or executable (the *emit* phase).
- Exposes a *semantic model* that provides symbol information (obtained from *binding*).

The semantic model is essential in implementing features such as symbol renaming, or offering code completion listings in an editor.

Creating a Compilation

Whether you're interested in querying the semantic model or performing a full compilation, the first step is to create a `CSharpCompilation`, passing in the (simple) name of the assembly that you want to create:

```
var compilation = CSharpCompilation.Create ("test");
```

An assembly's simple name is important even if you don't plan to emit an assembly, because it forms part of the identity of the types inside the compilation.

By default, it assumes that you want to create a library. You can specify a different kind of output (windows executable, console executable, etc.), as follows:

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication));
```

The `CSharpCompilationOptions` class has more than a dozen optional constructor parameters for options that you can pass to the compiler. For example, to enable compiler optimizations, you would do this:

```
compilation = compilation.WithOptions (
  new CSharpCompilationOptions (OutputKind.ConsoleApplication,
    optimizationLevel:OptimizationLevel.Release));
```

Next, let's add syntax trees. Each syntax tree corresponds to a "file" to be included in the compilation:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (""Hello"");
}");

compilation = compilation.AddSyntaxTrees (tree);
```

Finally, we need to reference the .NET runtime assemblies. Because it's difficult to know exactly what combination of assemblies are required, it's easiest to reference them all. The following code returns all the .NET runtime assemblies (plus any that the calling application references):

```
string trustedAssemblies = (string)AppContext.GetData
                           ("TRUSTED_PLATFORM_ASSEMBLIES");
string[] trustedAssemblyPaths = trustedAssemblies.Split(Path.PathSeparator);
```

> Note that this returns *runtime assemblies*, which are specific to the current platform and .NET version. If you're planning to use Roslyn to compile libraries that will work correctly across different platforms and .NET versions, you should use *reference assemblies*, instead. The reference assemblies are available in the NuGet package *Microsoft.NETCore.app.ref* (for .NET Core),

Microsoft.AspNetCore.App.ref (for ASP.NET Core) and *Microsoft.WindowsDesktop.app.ref* (for Windows Forms/WPF).

We then can add the references to the compilation, as follows:

```
var references = trustedAssemblyPaths.Select
  (path => MetadataReference.CreateFromFile (path));

compilation = compilation.AddReferences (references);
```

The call to `MetadataReference.CreateFromFile` reads the content of an assembly into memory, but not using ordinary reflection. Instead, it uses a high-performance assembly reader (*System.Reflection.Metadata*), which avoids creating an `Assembly` object. (Creating an `Assembly` object would be slow, and result in the assembly file being locked until the process exited).

The `PortableExecutableReference` that you get back from `MetadataReference.CreateFromFile` can end up with a significant memory footprint, so be careful about holding onto references that you don't need. Also, if you find yourself repeatedly creating references to the same assembly, a cache is worth considering (one that holds weak references is ideal).

You can do everything in a single step by calling the overload of `CSharpCompilation.Create` that takes syntax trees, references and options. Or you can do it fluently in a single expression, too:

```
var compilation = CSharpCompilation.Create ("...")
  .WithOptions (...)
  .AddSyntaxTrees (...)
  .AddReferences (...);
```

Diagnostics

A compilation can generate errors and warnings even if the syntax trees are error free. Examples include forgetting to import a namespace, a typo when referring to a type or member name, and type parameter inference failing. You can get the errors and warnings by calling `GetDiagnostics` on the compilation object. Any syntax errors will be included, too.

Emitting an Assembly

Creating an output assembly is simply a matter of calling `Emit`:

```
EmitResult result = compilation.Emit (@"c:\temp\test.dll");
Console.WriteLine (result.Success);
```

If `result.Success` is false, `EmitResult` also has a `Diagnostics` property to indicate the errors that occurred during emission (this also includes diagnostics from the previous stages). If `Emit` fails due to a file I/O error, it will throw an exception rather than generate error codes.

With .NET 5+ and .NET Core, you must specify a *.dll* extension even for Console or Windows applications. To run the application, you then call *dotnet.exe* with the path to your *.dll*.

The `Emit` method also lets you specify a *.pdb* file path (for debug information), and an XML documentation file path.

Querying the Semantic Model

Calling `GetSemanticModel` on a compilation returns the *semantic model* for a syntax tree:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main() => System.Console.WriteLine (123);
}");
```

```
var references = ((string)AppContext.GetData("TRUSTED_PLATFORM_ASSEMBLIES"))
  .Split (Path.PathSeparator)
  .Select (path => MetadataReference.CreateFromFile (path));

var compilation = CSharpCompilation.Create ("test")
  .AddReferences (references)
  .AddSyntaxTrees (tree);

SemanticModel model = compilation.GetSemanticModel (tree);
```

(The reason for needing to specify a tree is that a compilation can contain multiple trees.)

You might expect a semantic model to be similar to syntax tree, but with more properties and methods and a more detailed structure. This is not the case and there is no overarching DOM associated with the semantic model. Instead, you're given set of methods to call to obtain semantic information about a particular position or node in the syntax tree.

This means that you can't "explore" a semantic model like you would a syntax tree, and using it is rather like playing "20 Questions": the challenge is figuring out the right questions to ask. There are nearly 50 methods and extension methods; in this section, we'll cover some of the most commonly used methods, in particular, those that demonstrate the principles of using the semantic model.

Following on from our previous example, we could ask for symbol information on the `WriteLine` identifier, as follows:

```
var writeLineNode = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "WriteLine").Parent;

SymbolInfo symbolInfo = model.GetSymbolInfo (writeLineNode);
Console.WriteLine (symbolInfo.Symbol);   // System.Console.WriteLine(int)
```

`SymbolInfo` is a wrapper for symbols, whose nuances we discuss shortly. We begin first with symbols.

Symbols

In the syntax tree, names such as `System`, `Console`, and `WriteLine` are parsed as *identifiers* (`IdentifierNameSyntax` node). Identifiers have little meaning, and the syntactic parser does no work on "understanding" them other than to distinguish them from contextual keywords.

The semantic model is able to transform identifiers into *symbols*, which have type information (the output of the *binding* phase).

All symbols implement the `ISymbol` interface, although there are more specific interfaces for each kind of symbol. In our example, `System`, `Console`, and `WriteLine` map to symbols of the following types:

```
System      INamespaceSymbol
Console     INamedTypeSymbol
WriteLine   IMethodSymbol
```

Some symbol types, such as `IMethodSymbol` have a conceptual analog in the `System.Reflection` namespace (`MethodInfo`, in this case), whereas some other symbol types, such as `INamespaceSymbol`, do not. This is because the Roslyn type system exists for the benefit of the compiler, whereas the Reflection type system exists for the benefit of the CLR (after the source code has melted away).

Nonetheless, working with `ISymbol` types is similar in many ways to using the Reflection API we described in Chapter 18. Let's extend our previous example:

```
ISymbol symbol = model.GetSymbolInfo (writeLineNode).Symbol;

Console.WriteLine (symbol.Name);                  // WriteLine
Console.WriteLine (symbol.Kind);                  // Method
Console.WriteLine (symbol.IsStatic);              // True
Console.WriteLine (symbol.ContainingType.Name);   // Console
```

```
var method = (IMethodSymbol) symbol;
Console.WriteLine (method.ReturnType.ToString());  // void
```

The output of the last line illustrates a subtle difference with Reflection. Notice that "void" is in lowercase, which is C# nomenclature (Reflection is language-agnostic). Similarly, calling `ToString()` on the `INamedTypeSymbol` for `System.Int32` returns "int". Here's something else you can't do with Reflection:

```
Console.WriteLine (symbol.Language);                // C#
```

> With the syntax trees API, the classes for syntax nodes differ for C# and Visual Basic (although they share an abstract `SyntaxNode` base type). This makes sense because the languages have a different lexical structure. In contrast, `ISymbol` and its derived interfaces are shared between C# and Visual Basic. However, their internal concrete implementations *are* specific to each language, and the output from their methods and properties reflects language-specific differences.

We can also ask the symbol where it came from:

```
var location = symbol.Locations.First();
Console.WriteLine (location.Kind);                      // MetadataFile
```

If the symbol was defined in our own source code (i.e., a syntax tree), the `SourceTree` property will return that tree, and `SourceSpan` will return its location in the tree:

```
Console.WriteLine (location.SourceTree == null);    // True
Console.WriteLine (location.SourceSpan);            // [0..0)
```

A partial type can have multiple definitions, in which case it will have multiple `Locations`.

The following query returns all the overloads of `WriteLine`:

```
symbol.ContainingType.GetMembers ("WriteLine").OfType<IMethodSymbol>()
```

You can also call `ToDisplayParts` on a symbol. This returns a collection of "parts" that make up the full name; in our case `System.Console.WriteLine(int)` comprises four symbols interspersed with punctuation.

SymbolInfo

If you're writing code completion for an editor, you'll need to obtain symbols for code that's incomplete or incorrect. For instance, consider the following incomplete code:

```
System.Console.Writeline(
```

Because the `WriteLine` method is overloaded, it's impossible to match to a single `ISymbol`. Instead, we want to present options to the user. To deal with this, the semantic model's `GetSymbolInfo` method returns an `ISymbolInfo` struct which has the following properties:

```
ISymbol Symbol
ImmutableArray<ISymbol> CandidateSymbols
CandidateReason CandidateReason
```

If there's an error or ambiguity, the `Symbol` property returns null, and `CandidateSymbols` returns a collection comprising the best matches. The `CandidateReason` property returns an enum telling you what went wrong.

> To obtain error and warning information for a section of code, you can also call `GetDiagnostics` on a semantic model, specifying a `TextSpan`. Calling `GetDiagnostics` with no argument is equivalent to calling the same method on the `CSharpCompilation` object.

Symbol accessibility

`ISymbol` has a `DeclaredAccessibility` property which indicates whether the symbol is public, protected, internal, and so on. However, this isn't sufficient to determine whether a given symbol is accessible at a particular position in your source code. Local variables, for instance, have a lexically limited scope, and a protected class

member is accessible from source code positions within its type or a derived type. To help with this, `SemanticModel` has an `IsAccessible` method:

```
bool canAccess = model.IsAccessible (42, someSymbol);
```

This returns true if `someSymbol` can be accessed at offset 42 in the source code.

Declared symbols

If you call `GetSymbolInfo` on a type or member declaration, you'll get no symbols back. For instance, suppose that we want the symbol for our `Main` method:

```
var mainMethod = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "Main").Parent;

SymbolInfo symbolInfo = model.GetSymbolInfo (mainMethod);
Console.WriteLine (symbolInfo.Symbol == null);              // True
Console.WriteLine (symbolInfo.CandidateSymbols.Length);     // 0
```

> This applies not just to type/member declarations, but any node where you're *introducing* a new symbol rather than *consuming* an existing symbol.

To obtain the symbol, we must instead call `GetDeclaredSymbol`:

```
ISymbol symbol = model.GetDeclaredSymbol (mainMethod);
```

Unlike `GetSymbolInfo`, `GetDeclaredSymbol` either succeeds or it doesn't. (If it fails, it will because it can't find a valid declaration node.)

To give another example, suppose that our `Main` method is as follows:

```
static void Main()
{
  int xyz = 123;
}
```

We can determine the type of `xyz`, as follows:

```
SyntaxNode variableDecl = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "xyz").Parent;

var local = (ILocalSymbol) model.GetDeclaredSymbol (variableDecl);
Console.WriteLine (local.Type.ToString());             // int
Console.WriteLine (local.Type.BaseType.ToString());    // System.ValueType
```

TypeInfo

Sometimes, you need type information about an expression or literal for which there's no explicit symbol. Consider the following:

```
var now = System.DateTime.Now;
System.Console.WriteLine (now - now);
```

To determine the type of `now - now`, we call `GetTypeInfo` on the semantic model:

```
SyntaxNode binaryExpr = tree.GetRoot().DescendantTokens().Single (
  t => t.Text == "-").Parent;

TypeInfo typeInfo = model.GetTypeInfo (binaryExpr);
```

`TypeInfo` has two properties, `Type` and `ConvertedType`. The latter indicates the type after any implicit conversions:

```
Console.WriteLine (typeInfo.Type);              // System.TimeSpan
Console.WriteLine (typeInfo.ConvertedType);     // object
```

Because `Console.WriteLine` is overloaded to accept an `object` but not a `TimeSpan`, an implicit conversion to object took place, which manifested in `typeInfo.ConvertedType`.

Looking up symbols

A powerful feature of the semantic model is the ability to ask for all symbols in scope at a particular point in the source code. The result is the basis for IntelliSense listings, when the user requests a list of available symbols.

To obtain the listing, simply call `LookupSymbols`, with the desired source code offset. Here's a complete example:

```
var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static void Main()
  {
    int x = 123, y = 234;

  }
}");

var references = ((string)AppContext.GetData ("TRUSTED_PLATFORM_ASSEMBLIES"))
  .Split (Path.PathSeparator)
  .Select (path => MetadataReference.CreateFromFile (path));

var compilation = CSharpCompilation.Create ("test")
  .AddReferences (references)
  .AddSyntaxTrees (tree);

SemanticModel model = compilation.GetSemanticModel (tree);

// Look for available symbols at start of 6th line:
int index = tree.GetText().Lines[5].Start;

foreach (ISymbol symbol in model.LookupSymbols (index))
  Console.WriteLine (symbol.ToString());
```

Here's the result:

```
y
x
Program.Main()
object.ToString()
object.Equals(object)
object.Equals(object, object)
object.ReferenceEquals(object, object)
object.GetHashCode()
object.GetType()
object.~Object()
object.MemberwiseClone()
Program
Microsoft
System
Windows
```

(If we imported the `System` namespace, we'd see hundreds more symbols, for types in that namespace.)

Example: Renaming a Symbol

To illustrate the features we've covered, let's write a method to rename a symbol, which is robust to the most common use cases; in particular:

- The symbol can be a type, member, local variable, range, or loop variable.

- You can specify the symbol from either its use or declaration.
- In the case of a class or struct, it will rename the static and instance constructors.
- In the case of a class, it will rename the finalizer (destructor).

For brevity, we omit some checks, such as ensuring that the new name is not already in use, and that the symbol isn't an edge-case for which the rename will fail. Our method will consider just a single syntax tree, and so will have the following signature:

```
public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
```

One obvious way to implement this is to subclass `CSharpSyntaxRewriter`. However, a more elegant and flexible approach is to have `RenameSymbol` call a lower-level method that returns the text spans to be renamed:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
```

This allows an editor to call `GetRenameSpans` directly and apply just the changes (within an Undo transaction), avoiding the loss of editor state that might otherwise result in replacing the entire text.

This makes `RenameSymbol` a relatively simple wrapper around `GetRenameSpans`. We can use `SourceText`'s `WithChanges` method to apply a sequence of text changes:

```
public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
{
  IEnumerable<TextSpan> renameSpans = GetRenameSpans (model, token);

  SourceText newSourceText = model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (span => new TextChange (span, newName))
               .OrderBy (tc => tc));

  return model.SyntaxTree.WithChangedText (newSourceText);
}
```

`WithChanges` throws an exception unless the changes are in order; this is why we called `OrderBy` on the latter.

Now we must write `GetRenameSpans`. The first step is to find the symbol corresponding to the token that we want to rename. The token can be part of either a declaration or usage, so we first call `GetSymbolInfo`, and if the result is null, we call `GetDeclaredSymbol`:

```
public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;

  ISymbol symbol = model.GetSymbolInfo (node).Symbol
                ?? model.GetDeclaredSymbol (node);

  if (symbol == null) return null;   // No symbol to rename.
```

Next, we need to find the symbol definitions. We can get this from the symbol's `Locations` property. (Our consideration of multiple locations makes us robust to the scenario of partial classes and methods, although for the former to be useful, we would need to expand the example to work with multiple syntax trees).

```
  var definitions =
    from location in symbol.Locations
    where location.SourceTree == node.SyntaxTree
    select location.SourceSpan;
```

Now we need to find usages of the symbol. For this, we begin by looking for descendant tokens whose names match the symbol's name because this is a fast way to weed out most tokens. Then, we can call `GetSymbolInfo` on the token's parent node and see whether it matches the symbol we want to rename:

```
  var usages =
    from t in model.SyntaxTree.GetRoot().DescendantTokens()
    where t.Text == symbol.Name
    let s = model.GetSymbolInfo (t.Parent).Symbol
    where s == symbol
    select t.Span;
```

> Binding-related operations such as asking for symbol information have a tendency to be slower than operations that consider just text or syntax trees. This is because the process of binding can require searching for types in assemblies, applying type inference rules, and checking for extensions methods.

If the symbol is something other than a named type (local variable, range variable, etc.), our job is done and we can return the definitions plus usages:

```
  if (symbol.Kind != SymbolKind.NamedType)
    return definitions.Concat (usages);
```

If the symbol is a named type, we need to rename its constructors and destructor, if present. To do so, we enumerate the descendant nodes, looking for type declarations whose names match the one we want to rename. Then, we get its *declared* symbol, and if it matches the one we're renaming, we locate its constructor and destructor methods, returning the spans of their identifiers if present:

```
  var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                           .OfType<TypeDeclarationSyntax>()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;

  return definitions.Concat (usages).Concat (structors);
}
```

Here's the complete listing, along with an example of how to use it:

```
void Demo()
{
  var tree = CSharpSyntaxTree.ParseText (@"class Program
{
  static Program() {}
  public Program() {}

  static void Main()
  {
    Program p = new Program();
    p.Foo();
  }

  void Foo() => Bar();
  void Bar() => Foo();
}
");

  var references = ((string)AppContext.GetData ("TRUSTED_PLATFORM_ASSEMBLIES"))
    .Split (Path.PathSeparator)
```

```
    .Select (path => MetadataReference.CreateFromFile (path));

  var compilation = CSharpCompilation.Create ("test")
    .AddReferences (references)
    .AddSyntaxTrees (tree);

  var model = compilation.GetSemanticModel (tree);

  var tokens = tree.GetRoot().DescendantTokens();

  // Rename the Program class to Program2:
  SyntaxToken program = tokens.First (t => t.Text == "Program");
  Console.WriteLine (RenameSymbol (model, program, "Program2").ToString());

  // Rename the Foo method to Foo2:
  SyntaxToken foo = tokens.Last (t => t.Text == "Foo");
  Console.WriteLine (RenameSymbol (model, foo, "Foo2").ToString());

  // Rename the p local variable to p2:
  SyntaxToken p = tokens.Last (t => t.Text == "p");
  Console.WriteLine (RenameSymbol (model, p, "p2").ToString());
}

public SyntaxTree RenameSymbol (SemanticModel model, SyntaxToken token,
                                string newName)
{
  IEnumerable<TextSpan> renameSpans =
    GetRenameSpans (model, token).OrderBy (s => s);

  SourceText newSourceText = model.SyntaxTree.GetText().WithChanges (
    renameSpans.Select (s => new TextChange (s, newName)));

  return model.SyntaxTree.WithChangedText (newSourceText);
}

public IEnumerable<TextSpan> GetRenameSpans (SemanticModel model,
                                             SyntaxToken token)
{
  var node = token.Parent;

  ISymbol symbol =
    model.GetSymbolInfo (node).Symbol ??
    model.GetDeclaredSymbol (node);

  if (symbol == null) return null;   // No symbol to rename.

  var definitions =
    from location in symbol.Locations
    where location.SourceTree == node.SyntaxTree
    select location.SourceSpan;

  var usages =
    from t in model.SyntaxTree.GetRoot().DescendantTokens ()
    where t.Text == symbol.Name
    let s = model.GetSymbolInfo (t.Parent).Symbol
    where s == symbol
    select t.Span;

  if (symbol.Kind != SymbolKind.NamedType)
```

```
    return definitions.Concat (usages);

  var structors =
    from type in model.SyntaxTree.GetRoot().DescendantNodes()
                                          .OfType<TypeDeclarationSyntax>()
    where type.Identifier.Text == symbol.Name
    let declaredSymbol = model.GetDeclaredSymbol (type)
    where declaredSymbol == symbol
    from method in type.Members
    let constructor = method as ConstructorDeclarationSyntax
    let destructor = method as DestructorDeclarationSyntax
    where constructor != null || destructor != null
    let identifier = constructor?.Identifier ?? destructor.Identifier
    select identifier.Span;

  return definitions.Concat (usages).Concat (structors);
}
```

Chapter 28

I/O with UWP

File I/O in UWP

UWP applications are restricted in terms of the directories and files that they can access. The easiest way to navigate the restrictions is to use the types in the `Windows.Storage` namespace, the two primary classes being `StorageFolder` and `StorageFile`.

> In Windows Runtime for Windows 8 and 8.1, you couldn't use `FileStream` or the `Directory`/`File` classes at all. This made it more difficult to write portable class libraries, so this restriction has been relaxed in UWP for Windows 10, although the limits on what directories and files you can access still apply.

The types described this section are WinRT types, which are a part of the operating system rather than the .NET runtime. UWP runs on .NET Core 2.2.

Working with Directories

The `StorageFolder` class represents a directory. You can obtain a `StorageFolder` via its static method `GetFolderFromPathAsync`, giving it a full path to the folder. However, given that UWP lets you access files only in certain locations, an easier approach is to obtain a `StorageFolder` via a helper property such as `ApplicationData.Current.TemporaryFolder`, which returns a temporary folder that's isolated to your application.

> We describe all of the approaches for obtaining directories and files to which your application has access in "Obtaining Directories and Files."

`StorageFolder` has the properties you'd expect (`Name`, `Path`, `DateCreated`, `DateModified`, `Attributes`, and so on), methods to delete/rename the folder (`DeleteAsync`/`RenameAsync`), and methods to list files and subfolders (`GetFilesAsync` and `GetFoldersAsync`).

As is evident from their names, the methods are asynchronous, returning an object that you can convert into a task with the `AsTask` extension method, or directly await. The following obtains a directory listing of all files in the application's temporary folder:

```
StorageFolder tempFolder = ApplicationData.Current.TemporaryFolder;
IReadOnlyList<StorageFile> files = await tempFolder.GetFilesAsync();
foreach (IStorageFile file in files)
  Debug.WriteLine (file.Name);
```

The `CreateFileQueryWithOptions` method lets you filter to a specific extension:

```
StorageFolder tempFolder = ApplicationData.Current.TemporaryFolder;
var queryOptions = new QueryOptions (CommonFileQuery.DefaultQuery,
                                     new[] { ".txt" });
var txtFiles = await tempFolder.CreateFileQueryWithOptions (queryOptions)
                               .GetFilesAsync();
foreach (StorageFile file in txtFiles)
  Debug.WriteLine (file.Name);
```

The `QueryOptions` class exposes properties to further control the search. For example, the `FolderDepth` property requests a recursive directory listing:

```
queryOptions.FolderDepth = FolderDepth.Deep;
```

Working with Files

`StorageFile` is the primary class for working with files. You can obtain an instance from a full path (to which you have permission) with the static `StorageFile.GetFileFromPathAsync` method, or from a relative path by calling `GetFileAsync` method on a `StorageFolder` (or `IStorageFolder`) object:

```
StorageFolder tempFolder = ApplicationData.Current.TemporaryFolder;
StorageFile file = await tempFolder.GetFileAsync ("foo.txt");
```

If the file does not exist, a `FileNotFoundException` is thrown at that point.

`StorageFile` has properties such as `Name`, `Path`, and so on, and methods for working with files, such as `Move`, `Rename`, `Copy`, and `Delete` (all `Async`). The `CopyAsync` method returns a `StorageFile` corresponding to the new file. There's also a `CopyAndReplaceAsync`, which accepts a target `StorageFile` object rather than a target name and folder.

`StorageFile` also exposes methods to open the file for reading/writing via .NET streams (`OpenStreamForReadAsync` and `OpenStreamForWriteAsync`). For example, the following creates and writes to a file called *test.txt* in the temporary folder:

```
StorageFolder tempFolder = ApplicationData.Current.TemporaryFolder;

StorageFile file = await tempFolder.CreateFileAsync
  ("test.txt", CreationCollisionOption.ReplaceExisting);

using (Stream stream = await file.OpenStreamForWriteAsync())
using (StreamWriter writer = new StreamWriter (stream))
  await writer.WriteLineAsync ("This is a test");
```

> If you don't specify `CreationCollisionOption.ReplaceExisting` and the file already exists, it will automatically append a number to the filename to make it unique.

The following reads back the file:

```
StorageFolder tempFolder = ApplicationData.Current.TemporaryFolder;
StorageFile file = await tempFolder.GetFileAsync ("test.txt");

using (var stream = await file.OpenStreamForReadAsync ())
using (StreamReader reader = new StreamReader (stream))
  Debug.WriteLine (await reader.ReadToEndAsync());
```

Obtaining Directories and Files

In this section, we describe all of the locations to which UWP apps can potentially read and write files, and how to obtain them.

"Isolated" storage

The following `ApplicationData` folders are all private to your app and support both reading and writing:

```
Windows.Storage.ApplicationData.Current.LocalFolder
Windows.Storage.ApplicationData.Current.RoamingFolder
Windows.Storage.ApplicationData.Current.TemporaryFolder
```

The following writes, reads, and then deletes a file in `LocalFolder`:

```
StorageFolder localFolder = ApplicationData.Current.LocalFolder;
var myFile = Path.Combine (localFolder.Path, "full.txt");
await File.WriteAllTextAsync (myFile, "My Data");
var data = await File.ReadAllTextAsync (myFile);
File.Delete (myFile);
```

Application folder

A UWP app has read-only access to the folder in which the application is installed. There are two ways to access this folder; the first is to use the `Package` class in the `Windows.ApplicationModel` namespace to obtain a `StorageFolder`:

```
StorageFolder installedLocation = Package.Current.InstalledLocation;
string txt = await File.ReadAllTextAsync (
  Path.Combine (installedLocation.Path, "test.txt"));
```

The second is to directly obtain a `StorageFile` with an *app URI*:

```
StorageFile file = await StorageFile.
  GetFileFromApplicationUriAsync (new Uri ("ms-appx:///test.txt"));

using (var st = await file.OpenStreamForReadAsync())
using (var tr = new StreamReader (st))
  Console.WriteLine (await tr.ReadToEndAsync());
```

KnownFolders

The `KnownFolders` class exposes a static property for each of the following (potentially) permitted locations:

```
public static StorageFolder AppCaptures { get; }
public static StorageFolder CameraRoll { get; }
public static StorageFolder DocumentsLibrary { get; }
public static StorageFolder HomeGroup { get; }
public static StorageFolder MediaServerDevices { get; }
public static StorageFolder MusicLibrary { get; }
public static StorageFolder Objects3D { get; }
public static StorageFolder PicturesLibrary { get; }
public static StorageFolder Playlists { get; }
public static StorageFolder RecordedCalls { get; }
public static StorageFolder RemovableDevices { get; }
public static StorageFolder SavedPictures { get; }
public static StorageFolder VideosLibrary { get; }
```

If you want to access any of these locations, you must declare them in the application's package manifest (in Visual Studio 2019, you can directly edit the manifest; in Solution Explorer, right-click the manifest file and then choose View Code):

```
<Capabilities>
  <Capability Name="internetClient" />
  <uap:Capability Name="documentsLibrary" />
</Capabilities>
```

In addition, UWP applications can access only those files whose extensions match their declared file type associations, which you can specify in Visual Studio 2019's manifest editor, on the Declarations tab.

`KnownFolders` also has properties for accessing removable devices and home group folders.

Removable devices

If your app uses the AutoPlay extension, it can access files on connected devices, if the file extension is declared in the application manifest.

Downloads folder

UWP apps can create files in the Downloads folder and have full access to the files created. However, you can do so only through the `StorageFile` instance; you cannot use methods such as `File.WriteAllTextAsync` or `File.Delete`:

```
StorageFile newFile = await DownloadsFolder.CreateFileAsync
                      ("MyDownload.txt");

using (var st = await newFile.OpenStreamForWriteAsync())
using (var tw = new StreamWriter (st))
    tw.Write ("My data");

using (var st = await newFile.OpenStreamForReadAsync())
using (var tr = new StreamReader (st))
{
    var txt = await tr.ReadToEndAsync();
    ...
}
await newFile.DeleteAsync();
```

User-selected files and folders

Your UWP application can also access any file or folder that the user explicitly chooses via a `FileOpenPicker` or `FolderPicker` dialog (subject to normal OS permissions for the underlying user).

Using a `FileOpenPicker`:

```
FileOpenPicker openPicker = new FileOpenPicker();
openPicker.ViewMode = PickerViewMode.Thumbnail;
openPicker.SuggestedStartLocation = PickerLocationId.Desktop;
openPicker.FileTypeFilter.Add (".txt");

StorageFile picked = await openPicker.PickSingleFileAsync();
if (picked != null)
{
  using (var st = await picked.OpenStreamForReadAsync())
  using (var sr = new StreamReader (st))
  {
    var txt = sr.ReadToEnd();
  }
}
```

Using a `FolderPicker`:

```
FolderPicker dirPicker = new FolderPicker();
dirPicker.ViewMode = PickerViewMode.Thumbnail;
dirPicker.SuggestedStartLocation = PickerLocationId.Desktop;
dirPicker.FileTypeFilter.Add (".txt");

StorageFolder userFolder = await dirPicker.PickSingleFolderAsync();
if (userFolder != null)
{
  var userFile = await userFolder.CreateFileAsync ("InUserFolder.txt");
  using (var st = await userFile.OpenStreamForWriteAsync())
```

```
  using (var sw = new StreamWriter (st))
    sw.Write ("My data file in user-picked folder.");

  using (var st = await userFile.OpenStreamForReadAsync())
  using (var sr = new StreamReader (st))
  {
    var txt = sr.ReadToEnd();
  }
  await userFile.DeleteAsync();
}
```

TCP in UWP

In UWP applications, TCP functionality is exposed through WinRT types in the `Windows.Networking.Sockets` namespace. As with the .NET implementation, there are two primary classes to handle server and client roles, `StreamSocketListener` and `StreamSocket`.

> Your application manifest must declare the capability *Internet (Client)* if the host is on the internet or *Private Networks (Client & Server)* if the host is private (including `localhost`).

The following method starts a server on port 51111, and waits for a client to connect. It then reads a single message comprising a length-prefixed string:

```
async void Server()
{
  var listener = new StreamSocketListener();
  listener.ConnectionReceived += async (sender, args) =>
  {
    using (StreamSocket socket = args.Socket)
    {
      var reader = new DataReader (socket.InputStream);
      await reader.LoadAsync (4);
      uint length = reader.ReadUInt32();
      await reader.LoadAsync (length);
      Debug.WriteLine (reader.ReadString (length));
    }
    listener.Dispose();   // Close listener after one message.
  };
  await listener.BindServiceNameAsync ("51111");
}
```

In this example, we used a WinRT type called `DataReader` (in `Windows.Networking`) to read from the input stream, rather than converting to a .NET `Stream` object and using a `BinaryReader`. `DataReader` is rather like `BinaryReader` except that it supports asynchrony. The `LoadAsync` method asynchronously reads a specified number of bytes into an internal buffer, which then allows you to call methods such as `ReadUInt32` or `ReadString`. The idea is that if you wanted to, say, read 1,000 integers in a row, you'd first call `LoadAsync` with a value of `4000`, and then `ReadInt32` 1,000 times in a loop. This avoids the overhead of calling asynchronous operations in a loop (because each asynchronous operation incurs a small overhead).

> `DataReader`/`DataWriter` have a `ByteOrder` property to control whether numbers are encoding in big- or little-endian format. Big-endian is the default.

The `StreamSocket` object that we obtained from awaiting `AcceptAsync` has separate input and output streams. So, to write a message back, we'd use the socket's `OutputStream`. We can illustrate the use of `OutputStream` and `DataWriter` with the corresponding client code:

```
async void Client()
{
  using (var socket = new StreamSocket())
  {
    await socket.ConnectAsync (new HostName ("localhost"), "51111",
                               SocketProtectionLevel.PlainSocket);
    var writer = new DataWriter (socket.OutputStream);
    string message = "Hello!";
    uint length = (uint) Encoding.UTF8.GetByteCount (message);
    writer.WriteUInt32 (length);
    writer.WriteString (message);
    await writer.StoreAsync();
  }
}
```

We start by directly instantiating a `StreamSocket` and then call `ConnectAsync` with the host name and port. (You can pass either a DNS name or an IP address string into `HostName`'s constructor.) By specifying `SocketProtectionLevel.Ssl`, you can request SSL encryption (if configured on the server).

Again, we used a WinRT `DataWriter` rather than a .NET `BinaryWriter` and wrote the length of the string (measured in bytes rather than characters), followed by the string itself which is UTF-8 encoded. Finally, we called `StoreAsync`, which writes the buffer to the backing stream, and closed the socket.