Abstracting away prefixes is usually exactly what you want. If necessary, you can see what prefix was used through the `Prefix` property and convert it into a namespace by calling `LookupNamespace`.

XmlWriter

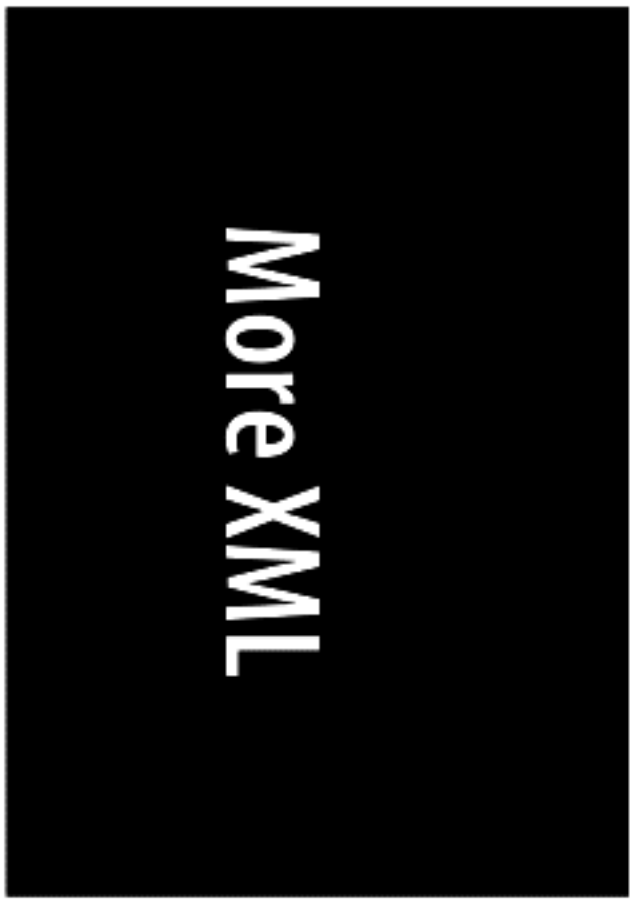
`XmlWriter` is a forward-only writer of an XML stream. The design of `XmlWriter` is symmetrical to `XmlReader`.

As with `XmlTextReader`, you construct an `XmlWriter` by calling `Create` with an optional `settings` object. In the following example, we enable indenting to make the output more human-readable, and then write a simple XML file:

```
XmlWriterSettings settings = new XmlWriterSettings();
settings.Indent = true;

using (XmlWriter writer = XmlWriter.Create ("..\\..\\foo.xml", settings))
{
  writer.WriteStartElement ("customer");
```

```
{
  writer.WriteStartElement ("customer");
  writer.WriteElementString ("firstname", "Jim");
  writer.WriteElementString ("lastname"," Bo");
  writer.WriteEndElement();
}
```



This produces the following document (the same as the file we read in the first example of `XmlReader`):

example of XmlReader):

```
<?xml version="1.0" encoding="utf-8" ?>
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

XmlWriter automatically writes the declaration at the top unless you indicate otherwise in XmlWriterSettings, by setting OmitXmlDeclaration to true or ConformanceLevel to Fragment. The latter also permits writing multiple root nodes—something that otherwise throws an exception.

The WriteValue method writes a single text node. It accepts both string and nonstring types such as bool and DateTime, internally calling XmlConvert to perform XML-compliant string conversions:

```
writer.WriteStartElement ("birthdate");
writer.WriteValue (DateTime.Now);
writer.WriteEndElement();
```

```
writer.WriteValue (DateTime.Now);
writer.WriteEndElement();
```

In contrast, if we call:

```
WriteElementString ("birthdate", DateTime.Now.ToString());
```

the result would be both non-XML-compliant and vulnerable to incorrect parsing.

`WriteString` is equivalent to calling `WriteValue` with a string. `XmlWriter` automatically escapes characters that would otherwise be illegal within an attribute or element, such as `&` `<` `>`, and extended Unicode characters.

Writing Attributes

You can write attributes immediately after writing a `start` element:

```
writer.WriteStartElement ("customer");
```

```
writer.WriteStartElement ("customer");
writer.WriteAttributeString ("id", "1");
writer.WriteAttributeString ("status", "archived");
```

To write nonstring values, `WriteEndAttribute`. call `WriteStartAttribute`, `WriteValue`, and then

Writing Other Node Types

Writing Other Node Types

`XmlWriter` also defines the following methods for writing other kinds of nodes:

```
WriteBase64        // for binary data
WriteBinHex        // for binary data
WriteCData
WriteComment
WriteDocType
WriteEntityRef
WriteProcessingInstruction
WriteRaw
WriteWhitespace
```

`WriteRaw` directly injects a string into the output stream. There is also a `WriteNode` method that accepts an `XmlReader`, echoing everything from the given `XmlReader`.

Namespaces and Prefixes

The overloads for the `Write*` methods allow you to associate an element or attribute

The overloads for the `Write*` methods allow you to associate an element or attribute with a namespace. Let's rewrite the contents of the XML file in our previous example. This time we will associate all the elements with the *http://oreilly.com* namespace, declaring the prefix `o` at the `customer` element:

```
writer.WriteStartElement ("o", "customer", "http://oreilly.com");
writer.WriteElementString ("o", "firstname", "http://oreilly.com", "Jim");
writer.WriteElementString ("o", "lastname", "http://oreilly.com", "Bo");
writer.WriteEndElement();
```

The output is now as follows:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<o:customer xmlns:o='http://oreilly.com'>
  <o:firstname>Jim</o:firstname>
  <o:lastname>Bo</o:lastname>
</o:customer>
```

Notice how for brevity `XmlWriter` omits the child element's namespace declarations when they are already declared by the parent element.

when they are already declared by the parent element.

Patterns for Using XmlReader/XmlWriter

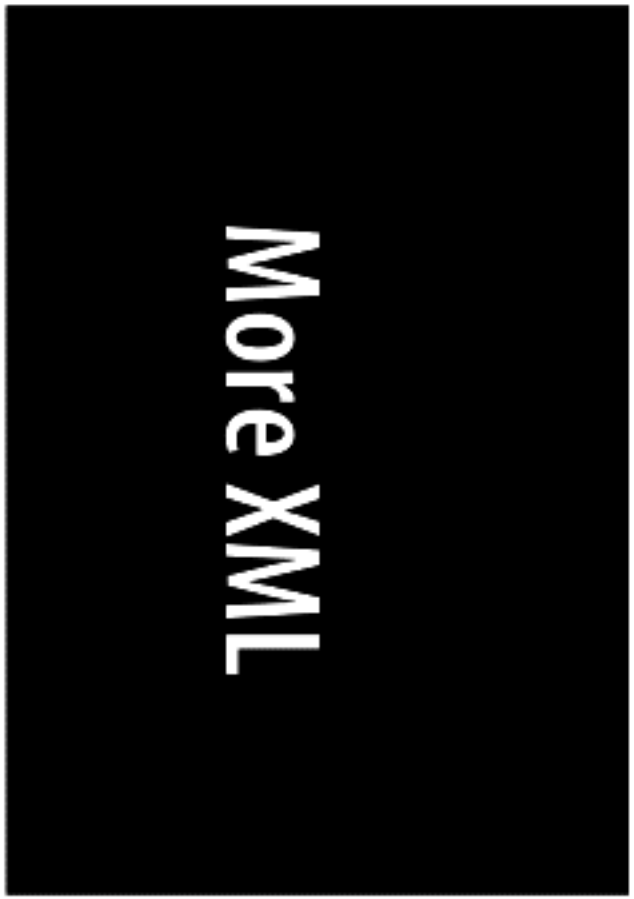
Working with Hierarchical Data

Consider the following classes:

```
public class Contacts
{
  public IList<Customer> Customers = new List<Customer>();
  public IList<Supplier> Suppliers = new List<Supplier>();
}

public class Customer { public string FirstName, LastName; }
```

```
public class Customer { public string FirstName, LastName; }
public class Supplier { public string Name;                 }
```



Suppose you want to use `XmlReader` and `XmlWriter` to serialize a `Contacts` object to XML as in the following:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<contacts>
```

```
<?xml version= 1.0  encoding= utf-8  standalone= yes ?>
<contacts>
   <customer id="1">
      <firstname>Jay</firstname>
      <lastname>Dee</lastname>
   </customer>
   <customer>                          <!-- we'll assume id is optional -->
      <firstname>Kay</firstname>
      <lastname>Gee</lastname>
   </customer>
   <supplier>
      <name>X Technologies Ltd</name>
   </supplier>
</contacts>
```

The best approach is not to write one big method, but to encapsulate XML functionality in the `Customer` and `Supplier` types themselves by writing `ReadXml` and `WriteXml` methods on these types. The pattern in doing so is straightforward:

-
-

`ReadXml` and `WriteXml` leave the reader/writer at the same depth when they exit.

`ReadXml` and `WriteXml` leave the reader/writer at the same depth when they exit. `ReadXml` reads the outer element, whereas `WriteXml` writes only its inner content.

Here's how we would write the `Customer` type:

```
public class Customer
{
  public const string XmlName = "customer";
  public int? ID;
  public string FirstName, LastName;

public Customer () { }
public Customer (XmlReader r) { ReadXml (r); }

public void ReadXml (XmlReader r)
{
  if (r.MoveToAttribute ("id")) ID = r.ReadContentAsInt();
```

```
{
  if (r.MoveToAttribute ("id")) ID = r.ReadContentAsInt();
  r.ReadStartElement();
```



```
  FirstName = r.ReadElementContentAsString ("firstname", "");
  LastName = r.ReadElementContentAsString ("lastname", "");
  r.ReadEndElement();
}

public void WriteXml (XmlWriter w)
{
  if (ID.HasValue) w.WriteAttributeString ("id", "", ID.ToString());
  w.WriteElementString ("firstname", FirstName);
  w.WriteElementString ("lastname", LastName);
}
```

Notice that `ReadXml` reads the outer start and end element nodes. If its caller did this job instead, `Customer` couldn't read its own attributes. The reason for not making `WriteXml` symmetrical in this regard is twofold:

job instead, `Customer` couldn't read its own attributes. The reason for not making `WriteXml` symmetrical in this regard is twofold:

- The caller might need to choose how the outer element is named.
- The caller might need to write extra XML attributes, such as the element's *subtype* (which could then be used to decide which class to instantiate when reading back the element).

Another benefit of following this pattern is that it makes your implementation compatible with `IXmlSerializable` (see Chapter 16).

The `Supplier` class is analogous:

```
public class Supplier
{
  public const string XmlName = "supplier";
```

```
public const string XmlName = "supplier";
public string Name;
```

```
public Supplier () { }
public Supplier (XmlReader r) { ReadXml (r); }

public void ReadXml (XmlReader r)
{
  r.ReadStartElement();
  Name = r.ReadElementContentAsString ("name", "");
  r.ReadEndElement();
}
```

```
public void WriteXml (XmlWriter w)
{
  w.WriteElementString ("name", Name);
}
```

```
    }
  }
```

With the `Contacts` class, we must enumerate the `customers` element in `ReadXml`, checking whether each subelement is a customer or a supplier. We also have to code around the empty element trap:

```
public void ReadXml (XmlReader r)
{
  bool isEmpty = r.IsEmptyElement;          // This ensures we don't get
  r.ReadStartElement();                     // snookered by an empty
```



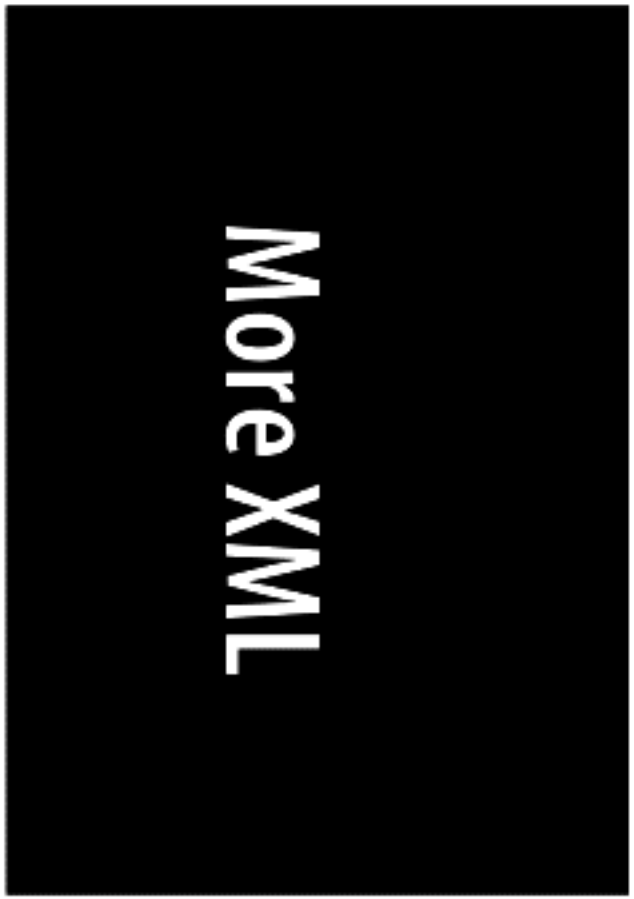
```
  if (isEmpty) return;                      // <contacts/> element!
  while (r.NodeType == XmlNodeType.Element)
  {
    if (r.Name == Customer.XmlName)      Customers.Add (new Customer (r));
    else if (r.Name == Supplier.XmlName) Suppliers.Add (new Supplier (r));
    else
      throw new XmlException ("Unexpected node: " + r.Name);
  }
```

```
    }
    r.ReadEndElement();
  }
```

```
  public void WriteXml (XmlWriter w)
  {
    foreach (Customer c in Customers)
    {
      w.WriteStartElement (Customer.XmlName);
      c.WriteXml (w);
      w.WriteEndElement();
    }
    foreach (Supplier s in Suppliers)
    {
      w.WriteStartElement (Supplier.XmlName);
      s.WriteXml (w);
      w.WriteEndElement();
    }
```

```
            w.WriteEndElement();
        }
    }
```

Mixing XmlReader/XmlWriter with an X-DOM



You can fly in an X-DOM at any point in the XML tree where `XmlReader` or `XmlWriter` becomes too cumbersome. Using the X-DOM to handle inner elements

You can fly in an X-DOM at any point in the XML tree where `XmlReader` or `XmlWriter` becomes too cumbersome. Using the X-DOM to handle inner elements is an excellent way to combine X-DOM's ease of use with the low-memory footprint of `XmlReader` and `XmlWriter`.

Using XmlReader with XElement

To read the current element into an X-DOM, you call `XNode.ReadFrom`, passing in the `XmlReader`. Unlike `XElement.Load`, this method is not "greedy" in that it doesn't expect to see a whole document. Instead, it reads just the end of the current subtree.

For instance, suppose we have an XML logfile structured as follows:

```
<log>
  <logentry id="1">
    <date>...</date>
    <source>...</source>
    ...
```

```
      <source>...</source>
      ...
    </logentry>
    ...
  </log>
```


If there were 1 million `logentry` elements, reading the whole thing into an X-DOM would waste memory. A better solution is to traverse each `logentry` with an `XmlReader`, and then use `XElement` to process the elements individually:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.IgnoreWhitespace = true;

using (XmlReader r = XmlReader.Create ("logfile.xml", settings))
{
  r.ReadStartElement ("log");
  while (r.Name == "logentry")
  {
```

```
  {
    XElement logEntry = (XElement) XNode.ReadFrom (r);
    int id = (int) logEntry.Attribute ("id");
    DateTime date = (DateTime) logEntry.Element ("date");
    string source = (string) logEntry.Element ("source");
    ...
  }
  r.ReadEndElement();
}
```

If you follow the pattern described in the previous section, you can slot an `XElement` into a custom type's `ReadXml` or `WriteXml` method without the caller ever knowing you've cheated! For instance, we could rewrite `Customer`'s `ReadXml` method as follows:

```
public void ReadXml (XmlReader r)
{
  XElement x = (XElement) XNode.ReadFrom (r);
  FirstName = (string) x.Element ("firstname");
  LastName = (string) x.Element ("lastname");
```

```
    FirstName = (string) x.Element ("firstname");
    LastName = (string) x.Element ("lastname");
  }
```

`XElement` collaborates with `XmlReader` to ensure that namespaces are kept intact and prefixes are properly expanded—even if defined at an outer level. So, if our XML file read like this:

```
<log xmlns="http://loggingspace">
  <logentry id="1">
    ...
```

the `XElements` we constructed at the `logentry` level would correctly inherit the outer namespace.

Using XmlWriter with XElement

You can use an `XElement` just to write inner elements to an `XmlWriter`. The following code writes 1 million `logentry` elements to an XML file using `XElement`—without storing the whole thing in memory:

storing the whole thing in memory:


```
using (XmlWriter w = XmlWriter.Create ("log.xml"))
{
  w.WriteStartElement ("log");
  for (int i = 0; i < 1000000; i++)
  {
    XElement e = new XElement ("logentry",
                   new XAttribute ("id", i),
                   new XElement ("date", DateTime.Today.AddDays (-1)),
                   new XElement ("source", "test"));
    e.WriteTo (w);
  }
  w.WriteEndElement ();
}
```

Using an `XElement` incurs minimal execution overhead. If we amend this example to use `XmlWriter` throughout, there's no measurable difference in execution time.

XmlDocument

`XmlDocument` is an in-memory representation of an XML document. Its object model and the methods that its types expose conform to a pattern defined by the W3C. So, if you're familiar with another W3C-compliant XML DOM (e.g., in Java), you'll be at home with `XmlDocument`. When compared to the X-DOM, however, the W3C model is much "clunkier."

at home with `XmlDocument`. When compared to the X-DOM, however, the W3C model is much "clunkier."

The base type for all objects in an `XmlDocument` tree is `XmlNode`. The following types derive from `XmlNode`:

```
XmlNode
   XmlDocument
   XmlDocumentFragment
   XmlEntity
   XmlNotation
   XmlLinkedNode
```

`XmlLinkedNode` exposes `NextSibling` and `PreviousSibling` properties and is an abstract base for the following subtypes:

```
XmlLinkedNode
```

```
XmlLinkedNode
    XmlCharacterData
    XmlDeclaration
    XmlDocumentType
    XmlElement
    XmlEntityReference
    XmlProcesingInstruction
```

Loading and Saving an XmlDocument

To load an `XmlDocument` from an existing source, you instantiate an `XmlDocument` and then call `Load` or `LoadXml`:

-
-

-
-

`Load` accepts a filename, `Stream`, `TextReader`, or `XmlReader`.

`LoadXml` accepts a literal XML string.

To save a document, call `Save` with a filename, `Stream`, `TextWriter`, or `XmlWriter`:

```
XmlDocument doc = new XmlDocument();
doc.Load ("customer1.xml");
doc.Save ("customer2.xml");
```

Traversing an XmlDocument

To illustrate traversing an `XmlDocument`, we'll use the following XML file:

To illustrate traversing an `XmlDocument`, we'll use the following XML file:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customer id="123" status="archived">
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

The `ChildNodes` property (defined in `XNode`) allows you to descend into the tree structure. This returns an indexable collection:

```
XmlDocument doc = new XmlDocument();
doc.Load ("customer.xml");

Console.WriteLine (doc.DocumentElement.ChildNodes[0].InnerText);
Console.WriteLine (doc.DocumentElement.ChildNodes[1].InnerText);

// Jim
```

```
// Jim
// Bo
```

With the `ParentNode` property, you can ascend back up the tree:

```
Console.WriteLine (
  doc.DocumentElement.ChildNodes[1].ParentNode.Name);
```

```
// customer
```

The following properties also help traverse the document (all of which return `null` if the node does not exist):

```
FirstChild
LastChild
NextSibling
PreviousSibling
```

PreviousSibling

The following two statements both output `firstname`:

```
Console.WriteLine (doc.DocumentElement.FirstChild.Name);
Console.WriteLine (doc.DocumentElement.LastChild.PreviousSibling.Name);
```

`XmlNode` exposes an `Attributes` property for accessing attributes either by name (and namespace) or by ordinal position. For example:

```
Console.WriteLine (doc.DocumentElement.Attributes ["id"].Value);
```

InnerText and InnerXml

The `InnerText` property represents the concatenation of all child text nodes. The following two lines both output `Jim`, since our XML document contains only a single text node:

```
Console.WriteLine (doc.DocumentElement.ChildNodes[0].InnerText);
Console.WriteLine (doc.DocumentElement.ChildNodes[0].FirstChild.Value);
```

Setting the InnerText property replaces *all* child nodes with a single text node. Be careful when setting InnerText to not accidentally wipe over element nodes. For example:

```
doc.DocumentElement.ChildNodes[0].InnerText = "Jo";              // wrong
doc.DocumentElement.ChildNodes[0].FirstChild.InnerText = "Jo";   // right
```

The InnerXml property represents the XML fragment *within* the current node. You typically use InnerXml on elements:

```
Console.WriteLine (doc.DocumentElement.InnerXml);
```

```
// OUTPUT:
<firstname>Jim</firstname><lastname>Bo</lastname>
```

`InnerXml` throws an exception if the node type cannot have children.

Creating and Manipulating Nodes

To create and add new nodes:

1. Call one of the `Create`*`XXX`* methods on the `XmlDocument`, such as `CreateElement`.
2. Add the new node into the tree by calling `AppendChild`, `PrependChild`, `InsertBefore`, or `InsertAfter` on the desired parent node.

InsertBefore, or InsertAfter on the desired parent node.

Creating nodes requires that you first have an XmlDocument—you cannot simply instantiate an XmlElement on its own like with the X-DOM. Nodes rely on a host XmlDocument for sustenance.

For example:

```
XmlDocument doc = new XmlDocument();
XmlElement customer = doc.CreateElement ("customer");
doc.AppendChild (customer);
```

The following creates a document matching the XML we started with earlier in this chapter in the section "XmlReader" on page 448:

```
XmlDocument doc = new XmlDocument ();
doc.AppendChild (doc.CreateXmlDeclaration ("1.0", null, "yes"));
```

```
XmlAttribute id     = doc.CreateAttribute ("id");
XmlAttribute status = doc.CreateAttribute ("status");
id.Value    = "123";
```

```
XmlAttribute status = doc.CreateAttribute ("status");
id.Value     = "123";
status.Value = "archived";

XmlElement firstname = doc.CreateElement ("firstname");
XmlElement lastname  = doc.CreateElement ("lastname");
firstname.AppendChild (doc.CreateTextNode ("Jim"));
lastname.AppendChild  (doc.CreateTextNode ("Bo"));

XmlElement customer = doc.CreateElement ("customer");
customer.Attributes.Append (id);
customer.Attributes.Append (status);
```



```
customer.AppendChild (lastname);
customer.AppendChild (firstname);
```

```
doc.AppendChild (customer);
```

You can construct the tree in any order. In the previous example, it doesn't matter if you rearrange the order of the lines that append child nodes.

To remove a node, you call `RemoveChild`, `ReplaceChild`, or `RemoveAll`.

Namespaces

See Chapter 10 for an introduction to XML namespaces and prefixes.

The `CreateElement` and `CreateAttribute` methods are overloaded to let you specify a namespace and prefix:

```
CreateXXX (string name);
CreateXXX (string name, string namespaceURI);
```

```
CreateXXX (string name);
CreateXXX (string name, string namespaceURI);
CreateXXX (string prefix, string localName, string namespaceURI);
```

The `name` parameter refers to either a local name (i.e., no prefix) or a name qualified with a prefix. The `namespaceURI` parameter is used if and only if you are *declaring* (rather than merely referring to) a namespace.

Here is an example of *declaring* a namespace with a prefix while creating an element:

```
XmlElement customer = doc.CreateElement ("o", "customer",
                                         "http://oreilly.com");
```

Here is an example of *referring* to a namespace with a prefix while creating an element:

```
XmlElement customer = doc.CreateElement ("o:firstname");
```

In the next section, we will explain how to deal with namespaces when writing XPath queries.

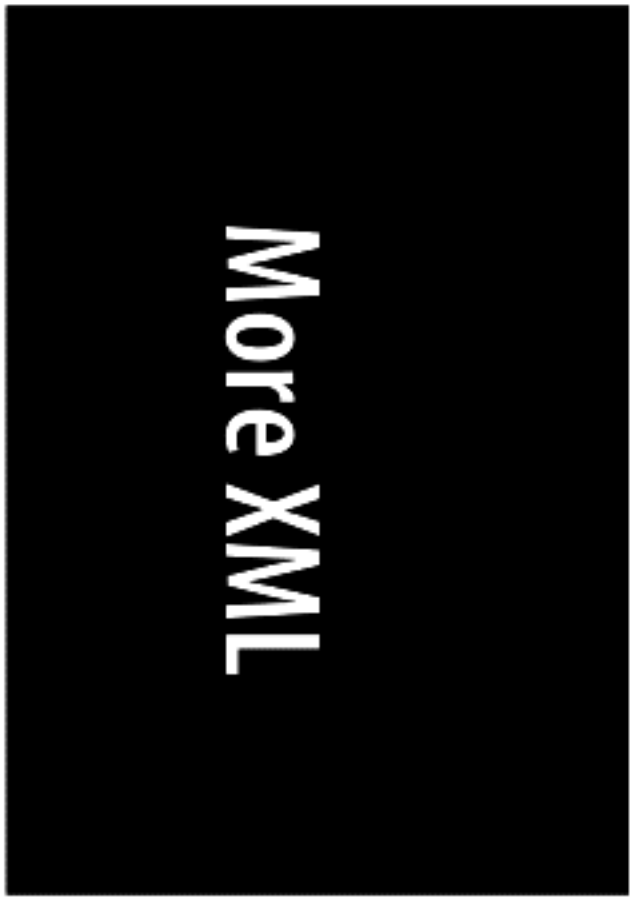
XPath

XPath is the W3C standard for XML querying. In the .NET Framework, XPath can query an `XmlDocument` rather like LINQ queries an X-DOM. XPath has a wider scope, though, in that it's also used by other XML technologies, such as XML schema, XLST, and XAML.

XPath queries are expressed in terms of the XPath 2.0 Data Model. Both the DOM and the XPath Data Model represent an XML document as a tree. The difference is that the XPath Data Model is purely data-centric, abstracting away the formatting aspects of XML text. For example, CDATA sections are not required in the XPath Data Model, since the only reason CDATA sections exist is to enable text to contain markup character sequences. The XPath specification is at *http://www.w3.org/tr/xpath20/*

quences. The XPath specification is at *http://www.w3.org/tr/xpath20/*.

The examples in this section all use the following XML file:




```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<customers>
  <customer id="123" status="archived">
```

```
<customers>
  <customer id="123" status="archived">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer>
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

You can write XPath queries within code in the following ways:

-
-
-

Call one of the `Select`*`XXX`* methods on an `XmlDocument` or `XmlNode`.

Spawn an `XPathNavigator` from either:

Spawn an XPathNavigator from either:

— An XmlDocument

— An XPathDocument

Call an XPath*XXX* extension method on an XNode.

The Select*XXX* methods accept an XPath query string. For example, the following finds the firstname node of an XmlDocument:

```
XmlDocument doc = new XmlDocument();
doc.Load ("customers.xml");
XmlNode n = doc.SelectSingleNode ("customers/customer[firstname='Jim']");
Console.WriteLine (n.InnerText);  // JimBo
```

The Select*XXX* methods delegate their implementation to XPathNavigator, which you can also use directly—over either an XmlDocument or a read-only XPathDocument.

You can also execute XPath queries over an X-DOM, via extension methods defined in System.Xml.XPath:

in System.Xml.XPath:

```
XDocument doc = XDocument.Load (@"Customers.xml");
XElement e = e.XPathSelectElement ("customers/customer[firstname='Jim']");
Console.WriteLine (e.Value);  // JimBo
```


The extension methods for use with XNodes are:

- CreateNavigator
- XPathEvaluate
- XPathSelectElement
- XPathSelectElements

Common XPath Operators

The XPath specification is huge. However, you can get by knowing just a few operators (see Table 11-2), just as you can play a lot of songs knowing just three chords.

Table 11-2. Common XPath operators

Table 11-2. Common XPath operators

Operator	Description
/	Children
//	Recursively children
.	Current node (usually implied)
..	Parent node
*	Wildcard
@	Attribute
[ ]	Filter
:	Namespace separator

:	Namespace separator

To find the `customers` node:

```
XmlNode node = doc.SelectSingleNode ("customers");
```

The / symbol queries child nodes. To select the `customer` nodes:

```
XmlNode node = doc.SelectSingleNode ("customers/customer");
```

The // operator includes all child nodes, regardless of nesting level. To select all `lastname` nodes:

```
XmlNodeList nodes = doc.SelectNodes ("//lastname");
```

The .. operator selects parent nodes. This example is a little silly because we're starting from the root anyway, but it serves to illustrate the functionality:

```
XmlNodeList nodes = doc.SelectNodes ("customers/customer..customers");
```

The * operator selects nodes regardless of name. The following selects the child nodes of `customer`, regardless of name:

The * operator selects nodes regardless of name. The following selects the child nodes of customer, regardless of name:

```
XmlNodeList nodes = doc.SelectNodes ("customers/customer/*");
```

The @ operator selects attributes. * can be used as a wildcard. Here is how to select the id attribute:

```
XmlNode node = doc.SelectSingleNode ("customers/customer/@id");
```


The [ ] operator filters a selection, in conjunction with the operators =, !=, <, >, not(), and, and or. In this example, we filter on firstname:

```
XmlNode n = doc.SelectSingleNode ("customers/customer[firstname='Jim']");
```

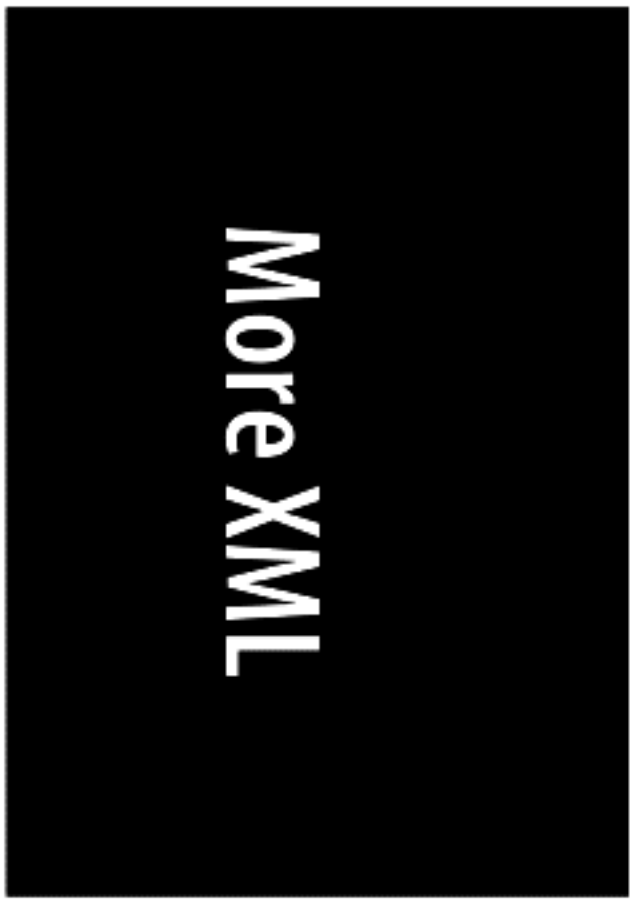
The : operator qualifies a namespace. Had the customers element been qualified with the x namespace, we would access it as follows:

```
XmlNode node = doc.SelectSingleNode ("x:customers");
```

XPathNavigator

XPathNavigator

`XPathNavigator` is a cursor over the XPath Data Model representation of an XML document. It is loaded with primitive methods that move the cursor around the tree (e.g., move to parent, move to first child, etc.). The `XPathNavigator`'s `Select*` methods take an XPath string to express more complex navigations or queries that return multiple nodes.



Spawn instances of `XPathNavigator` from an `XmlDocument`, an `XPathDocument`, or another `XPathNavigator`. Here is an example of spawning an `XPathNavigator` from an `XmlDoument`:

```
XPathNavigator nav = doc.CreateNavigator();
XPathNavigator jim = nav.SelectSingleNode
  (
    "customers/customer[firstname='Jim']"
  );
Console.WriteLine (jim.Value);

// JimBo
```

In the XPath Data Model, the value of a node is the concatenation of the text elements, equivalent to `XmlDocument`'s `InnerText` property.

ments, equivalent to `XmlDocument`'s `InnerText` property.

The `SelectSingleNode` method returns a single `XPathNavigator`. The `Select` method returns an `XPathNodeIterator`, which simply iterates over multiple `XPathNavigators`. For example:

```
XPathNavigator nav = doc.CreateNavigator();
string xPath = "customers/customer/firstname/text()";
foreach (XPathNavigator navC in nav.Select (xPath))
  Console.WriteLine (navC.Value);
```

OUTPUT:

```
Jim
Thomas
```

To perform faster queries, you can compile an XPath query into an `XPathExpression`. You then pass the compiled expression to a `Select*` method, instead of a string. For example:

```
XPathNavigator nav = doc.CreateNavigator();
XPathExpression expr = nav.Compile ("customers/customer/firstname");
foreach (XPathNavigator a in nav.Select (expr))
  Console.WriteLine (a.Value);
```

```
foreach (XPathNavigator a in nav.Select (expr))
  Console.WriteLine (a.Value);
```



```
OUTPUT:
Jim
Thomas
```

Querying with Namespaces

Querying elements and attributes that contain namespaces requires some extra unintuitive steps. Consider the following XML file:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>

<o:customers xmlns:o='http://oreilly.com'>
  <o:customer id="123" status="archived">
```

```
<o:customers xmlns:o="http://oreilly.com">
  <o:customer id="123" status="archived">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </o:customer>
  <o:customer>
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </o:customer>
</o:customers>
```

The following query will fail, despite qualifying the nodes with the prefix o:

```
XmlDocument doc = new XmlDocument();
doc.Load ("customers.xml");
XmlNode n = doc.SelectSingleNode ("o:customers/o:customer");
Console.WriteLine (n.InnerText);  // JimBo
```

To make this query work, you must first create an `XmlNamespaceManager` instance as follows:

```
XmlNamespaceManager xnm = new XmlNamespaceManager (doc.NameTable);
```

You can treat `NameTable` as a black box (`XmlNamespaceManager` uses it internally to cache and reuse strings). Once we create the namespace manager, we can add prefix/namespace pairs to it as follows:

```
xnm.AddNamespace ("o", "http://oreilly.com");
```

The `Select*` methods on `XmlDocument` and `XPathNavigator` have overloads that accept an `XmlNamespaceManager`. We can successfully rewrite the previous query as follows:

```
XmlNode n = doc.SelectSingleNode ("o:customers/o:customer", xnm);
```

XPathDocument

`XPathDocument` is used for read-only XML documents that conform to the W3C XPath Data Model. An `XPathNavigator` backed by an `XPathDocument` is faster than an `XmlDocument`, but it cannot make changes to the underlying document:

XmlDocument, but it cannot make changes to the underlying document:

```
XPathDocument doc = new XPathDocument ("customers.xml");
XPathNavigator nav = doc.CreateNavigator();
foreach (XPathNavigator a in nav.Select ("customers/customer/firstname"))
  Console.WriteLine (a.Value);
```

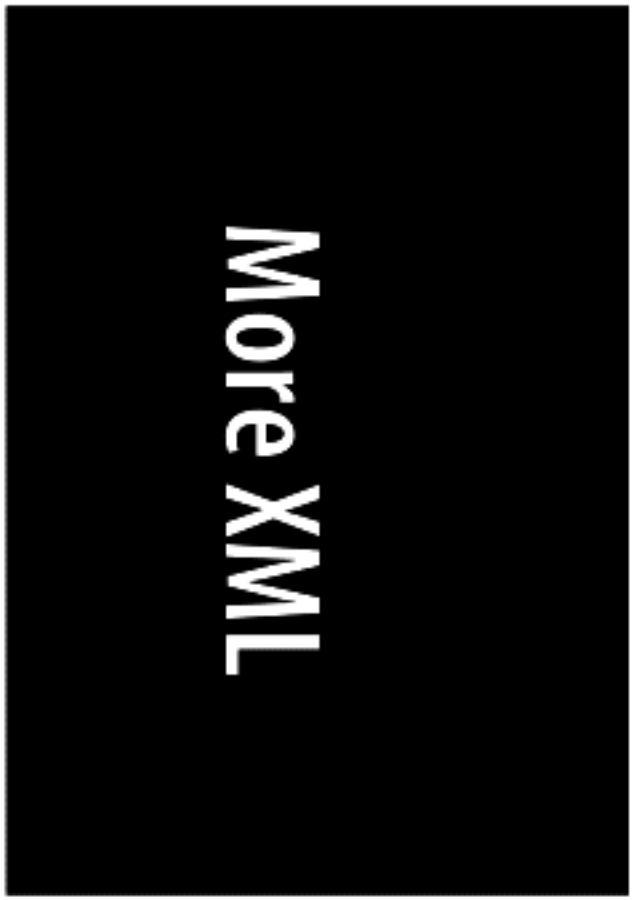

OUTPUT:

```
Jim
Thomas
```

XSD and Schema Validation

The content of a particular XML document is nearly always domain-specific, such as a Microsoft Word document, an application configuration document, or a web service. For each domain, the XML file conforms to a particular pattern. There are several standards for describing the schema of such a pattern, to standardize and automate the interpretation and validation of XML documents. The most widely accepted standard is XSD, short for *XML Schema Definition*. Its precursors, DTD

automate the interpretation and validation of XML documents. The most widely accepted standard is *XSD*, short for *XML Schema Definition*. Its precursors, DTD and XDR, are also supported by `System.Xml`.



Consider the following XML document:

```
<?xml version="1.0"?>
```

```
<?xml version="1.0"?>
<customers>
  <customer id="1" status="active">
    <firstname>Jim</firstname>
    <lastname>Bo</lastname>
  </customer>
  <customer id="1" status="archived">
    <firstname>Thomas</firstname>
    <lastname>Jefferson</lastname>
  </customer>
</customers>
```

We can write an XSD for this document as follows:

We can write an XSD for this document as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified"
           elementFormDefault="qualified"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="customers">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" name="customer">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="firstname" type="xs:string" />
              <xs:element name="lastname" type="xs:string" />
            </xs:sequence>
            <xs:attribute name="id" type="xs:int" use="required" />
            <xs:attribute name="status" type="xs:string" use="required" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

As you can see, XSD documents are themselves written in XML. Furthermore, an

As you can see, XSD documents are themselves written in XML. Furthermore, an XSD document is describable with XSD—you can find that definition at *http://www.w3.org/2001/xmlschema.xsd*.

Performing Schema Validation

You can validate an XML file or document against one or more schemas before reading or processing it. There are a number of reasons to do so:

-
-
-

You can get away with less error checking and exception handling.

Schema validation picks up errors you might otherwise overlook.

Error messages are detailed and informative.

Error messages are detailed and informative.

To perform validation, plug a schema into an `XmlReader`, an `XmlDocument`, or an XDOM object, and then read or load the XML as you would normally. Schema validation happens automatically as content is read, so the input stream is not read twice.

Validating with an XmlReader

Here's how to plug a schema from the file *customers.xsd* into an `XmlReader`:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  ...
```

If the schema is inline, set the following flag instead of adding to `Schemas`:

```
settings.ValidationFlags |= XmlSchemaValidationFlags.ProcessInlineSchema;
```

```
settings.ValidationFlags |= XmlSchemaValidationFlags.ProcessInlineSchema;
```

You then `Read` as you would normally. If schema validation fails at any point, an `XmlSchemaValidationException` is thrown.

Calling `Read` on its own validates both elements and attributes: you don't need to navigate to each individual attribute for it to be validated.

If you want *only* to validate the document, you can do this:

```
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { while (r.Read()) ; }
  catch (XmlSchemaValidationException ex)
  {
    ...
  }
```

`XmlSchemaValidationException` has properties for the error `Message`, `LineNumber`, and `LinePosition`. In this case, it only tells you about the first error in the document. If you want to report on all errors in the document, you instead must handle the `ValidationEventHandler` event:

you want to report on all errors in the document, you instead must handle the ValidationEventHandler event:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");
```



```
settings.ValidationEventHandler += ValidationHandler;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  while (r.Read()) ;
```

When you handle this event, schema errors no longer throw exceptions. Instead, they fire your event handler:

```
static void ValidationHandler (object sender, ValidationEventArgs e)
{
  Console.WriteLine ("Error: " + e.Exception.Message);
}
```

The Exception property of ValidationEventArgs contains the XmlSchemaValidationException that would have otherwise been thrown.

The `Exception` property of `ValidationEventArgs` contains the `XmlSchemaValidationException` that would have otherwise been thrown.

The `System.Xml` namespace also contains a class called `XmlValidatingReader`. This was used to perform schema validation prior to Framework 2.0, and it is now deprecated.

Validating an X-DOM or XmlDocument

To validate an XML file or stream while reading into an X-DOM or `XmlDocument`, you create an `XmlReader`, plug in the schemas, and then use the reader to load the DOM:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ValidationType = ValidationType.Schema;
settings.Schemas.Add (null, "customers.xsd");

XDocument doc;
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { doc = XDocument.Load (r); }
  catch (XmlSchemaValidationException ex) { ... }

XmlDocument xmlDoc = new XmlDocument();
using (XmlReader r = XmlReader.Create ("customers.xml", settings))
  try { xmlDoc.Load (r); }
  catch (XmlSchemaValidationException ex) { ... }
```

```
try { xmlDoc.Load (r); }
catch (XmlSchemaValidationException ex) { ... }
```

You can also validate an `XDocument` or `XElement` that's already in memory, by calling extension methods in `System.Xml.Schema`. These methods accept an `XmlSchemaSet` (a collection of schemas) and a validation event handler:

```
XDocument doc = XDocument.Load (@"customers.xml");
XmlSchemaSet set = new XmlSchemaSet ();
set.Add (null, @"customers.xsd");
StringBuilder errors = new StringBuilder ();
doc.Validate (set, (sender, args) => { errors.AppendLine
                                       (args.Exception.Message); }
              );
Console.WriteLine (errors.ToString());
```


To validate an `XmlDocument` already in memory, add the schema(s) to the `XmlDocument`'s `Schemas` collection and then call the document's `Validate` method, passing in a `ValidationEventHandler` to process the errors.

passing in a `ValidationEventHandler` to process the errors.

XSLT

XSLT stands for *Extensible Stylesheet Language Transformations*. It is an XML language that describes how to transform one XML language into another. The quintessential example of such a transformation is transforming an XML document (that typically describes data) into an XHTML document (that describes a formatted document).

Consider the following XML file:

```
<customer>
  <firstname>Jim</firstname>
  <lastname>Bo</lastname>
</customer>
```

</customer>

The following XSLT file describes such a transformation:

```
<?xml version="1.0" encoding="UTF-8"?>
  <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
  <xsl:template match="/">
    <html>
      <p><xsl:value-of select="//firstname"/></p>
      <p><xsl:value-of select="//lastname"/></p>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The output is as follows:

```
<html>
  <p>Jim</p>
```

```
    <p>Jim</p>
    <p>Bo</p>
</html>
```

The `System.Xml.Xsl.XslCompiledTransform` transform class efficiently performs XLST transforms. It renders `XmlTransform` obsolete. `XmlTransform` works very simply:

```
XslCompiledTransform transform = new XslCompiledTransform();
transform.Load ("test.xslt");
transform.Transform ("input.xml", "output.xml");
```

Generally, it's more useful to use the overload of `Transform` that accepts an `XmlWriter` rather than an output file, so you can control the formatting.

1

12

Disposal and Garbage Collection

Some objects require explicit tear-down code to release resources such as open files

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

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

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

IDisposable, Dispose, and Close

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

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

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

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

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


The compiler converts this to:

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

The compiler converts this to:

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

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

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

```
sealed class Demo : IDisposable
{
  public void Dispose()
```

```
    public void Dispose()
    {
      // Perform cleanup / tear-down.
      ...
    }
}
```

This pattern works well in simple cases and is appropriate for sealed classes. We'll describe a more elaborate pattern that can provide a backup for consumers that forget to call `Dispose` in "Calling Dispose from a Finalizer" on page 484. With unsealed types, there's a strong case for following this latter pattern from the outset—otherwise, it becomes very messy if the subtype wants to add such functionality itself.

Standard Disposal Semantics

Standard Disposal Semantics

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

1. Once disposed, an object is beyond redemption. It cannot be reactivated, and calling its methods or properties throws an `ObjectDisposedException`.
2. Calling an object's `Dispose` method repeatedly causes no error.
3. If disposable object *x* contains or "wraps" or "possesses" disposable object *y*, *x*'s `Dispose` method automatically calls *y*'s `Dispose` method—unless instructed otherwise.

* In "Interrupt and Abort" on page 855 in Chapter 21, we describe how aborting a thread can violate the safety of this pattern. This is rarely an issue in practice because aborting threads is widely discouraged for precisely this (and other) reasons.

These rules are also helpful when writing your own types, though not mandatory.

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

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

Close and Stop

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

-
-

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

An example of the latter is `IDbConnection`: a `Closed` connection can be re-`Open`ed; a `Disposed` connection cannot. Another example is a Windows `Form` activated with

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

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

When to Dispose

A safe rule to follow (in nearly all cases) is "If in doubt, dispose." A disposable object—if it could talk—would say the following:

> When you've finished with me, let me know. If simply abandoned, I might cause trouble for other object instances, the application domain, the computer, the network, or the database!

Objects wrapping an unmanaged resource handle will nearly always require disposal, in order to free the handle. Examples include Windows Forms controls, file or network streams, network sockets, GDI+ pens, brushes, and bitmaps. Conversely, if a type is disposable, it will often (but not always) reference an unmanaged handle, directly or indirectly. This is because unmanaged handles provide the gate-

versely, if a type is disposable, it will often (but not always) reference an unmanaged handle, directly or indirectly. This is because unmanaged handles provide the gateway to the "outside world" of operating system resources, network connections, database locks—the primary means by which objects can create trouble outside of themselves if improperly abandoned.

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

- When obtaining a *shared* object via a static field or property
- When an object's `Dispose` method does something that you don't want

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

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

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

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

(LINQ to SQL)	to that context

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

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

Opt-in Disposal

Because `IDisposable` makes a type tractable with C#'s `using` construct, there's a temptation to extend the reach of `IDisposable` to nonessential activities. For instance:

instance:

```
public sealed class HouseManager : IDisposable
{
  public void Dispose()
  {
    CheckTheMail();
  }
  ...
}
```


The idea is that a consumer of this class can choose to circumvent the nonessential cleanup—simply by not calling `Dispose`. This, however, relies on the consumer knowing what's inside `Demo`'s `Dispose` method. It also breaks if *essential* cleanup activity is later added:

```
public void Dispose()
{
```

```
public void Dispose()
{
  CheckTheMail();    // Nonessential
  LockTheHouse();    // Essential
}
```

The solution to this problem is the opt-in disposal pattern:

```
public sealed class HouseManager : IDisposable
{
  public readonly bool CheckMailOnDispose;

public Demo (bool checkMailOnDispose)
{
  CheckMailOnDispose = checkMailOnDispose;
}
```

Disposal and GC

```
public void Dispose()
{
    if (CheckMailOnDispose) CheckTheMail();
    LockTheHouse();
}
```

```
    ...
}
```

The consumer can then always call `Dispose`—providing simplicity and avoiding the need for special documentation or reflection. An example of where this pattern is implemented is in the `DeflateStream` class, in `System.IO.Compression`. Here's its constructor:

```
public DeflateStream (Stream stream, CompressionMode mode, bool leaveOpen)
```

The nonessential activity is closing the inner `stream` (the first parameter) upon disposal. There are times when you want to leave the inner stream open and yet still dispose the `DeflateStream` to perform its *essential* tear-down activity (flushing buffered data).

This pattern might look simple, yet it escaped `StreamReader` and `StreamWriter` (the `System.IO` namespace). The result is messy: `StreamWriter` must expose another method (`Flush`) to perform essential cleanup for consumers not calling `Dispose`. The `CryptoStream` class in `System.Security.Cryptography` suffers a similar problem and requires that you call `FlushFinalBlock` to tear it down while keeping the inner stream

`CryptoStream` class in `System.Security.Cryptography` suffers a similar problem and requires that you call `FlushFinalBlock` to tear it down while keeping the inner stream open.

You could describe this as an *ownership* issue. The question for a disposable object is: do I really own the underlying resource that I'm using? Or am I just renting it from someone else who manages both the underlying resource lifetime and, by some undocumented contract, my lifetime?

Following the opt-in pattern avoids this problem by making the ownership contract documented and explicit.

Clearing Fields in Disposal

In general, you don't need to clear an object's fields in its `Dispose` method. However, it is good practice to unsubscribe from events that the object has subscribed to in-

In general, you don't need to clear an object's fields in its `Dispose` method. However, it is good practice to unsubscribe from events that the object has subscribed to internally over its lifetime (see "Managed Memory Leaks" on page 491 for an example). Unsubscribing from such events avoids receiving unwanted event notifications—and avoids unintentionally keeping the object alive in the eyes of the garbage collector (GC).

A `Dispose` method itself does not cause memory to be released—this can happen only in garbage collection.

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

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

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

Occasionally, an object holds high-value secrets, such as encryption keys. In these

Occasionally, an object holds high-value secrets, such as encryption keys. In these cases, it can make sense to clear such data from fields during disposal (to avoid discovery by less privileged assemblies or malware). The `SymmetricAlgorithm` class in `System.Security.Cryptography` does exactly this, by calling `Array.Clear` on the byte array holding the encryption key.

Automatic Garbage Collection

Regardless of whether an object requires a `Dispose` method for custom tear-down logic, at some point the memory it occupies on the heap must be freed. The CLR handles this side of it entirely automatically, via an automatic GC. You never deallocate managed memory yourself. For example, consider the following method:


```
public void Test()
{
```

```
{
  byte[] myArray = new byte[1000];
  ...
}
```

Garbage Collection and Memory Consumption

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

The problem can look worse than it is, though, if you judge memory consumption by the "Memory Usage" figure reported by the Task Manager in Windows XP. Unlike with later versions of Windows (which report *private working set*), the XP figure includes memory that a process has internally deallocated and is willing to rescind immediately to the operating system should another process need it. (It doesn't return the memory to the operating system immediately to avoid the overhead of asking for it back, should it be required a short while later. It reasons: "If the computer has plenty of free memory, why not use it to lessen allocation/deallocation overhead?")

location overhead?")

You can determine your process's real memory consumption by querying a performance counter (`System.Diagnostics`):

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

Reading performance counters requires administrative privileges.

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

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

Garbage collection does not happen immediately after an object is orphaned. Rather like garbage collection on the street, it happens periodically, although (unlike garbage collection on the street) not to a fixed schedule. The CLR bases its decision on when to collect upon a number of factors, such as the available memory, the amount of memory allocation, and the time since the last collection. This means that there's

of memory allocation, and the time since the last collection. This means that there's

an indeterminate delay between an object being orphaned and being released from memory. Theoretically, it can range from nanoseconds to days.

The GC doesn't collect all garbage with every collection. Instead, the memory manager divides objects into *generations* and the GC collects new generations (recently allocated objects) more frequently than old generations (long-lived objects). We'll discuss this in more detail in "How the Garbage Collector Works" on page 487.

Roots

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

A root is one of the following:

-
-
-

A local variable or parameter in an executing method (or in any method in its call stack)

A static variable

An object on the queue that stores objects ready for finalization (see the next section)

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

Note that a group of objects that reference each other cyclically are considered dead without a root referee (see Figure 12-1). To put it in another way, objects that cannot be accessed by following the arrows (references) from a root object are

be accessed by following the arrows (references) from a root object are *unreachable*—and therefore subject to collection.

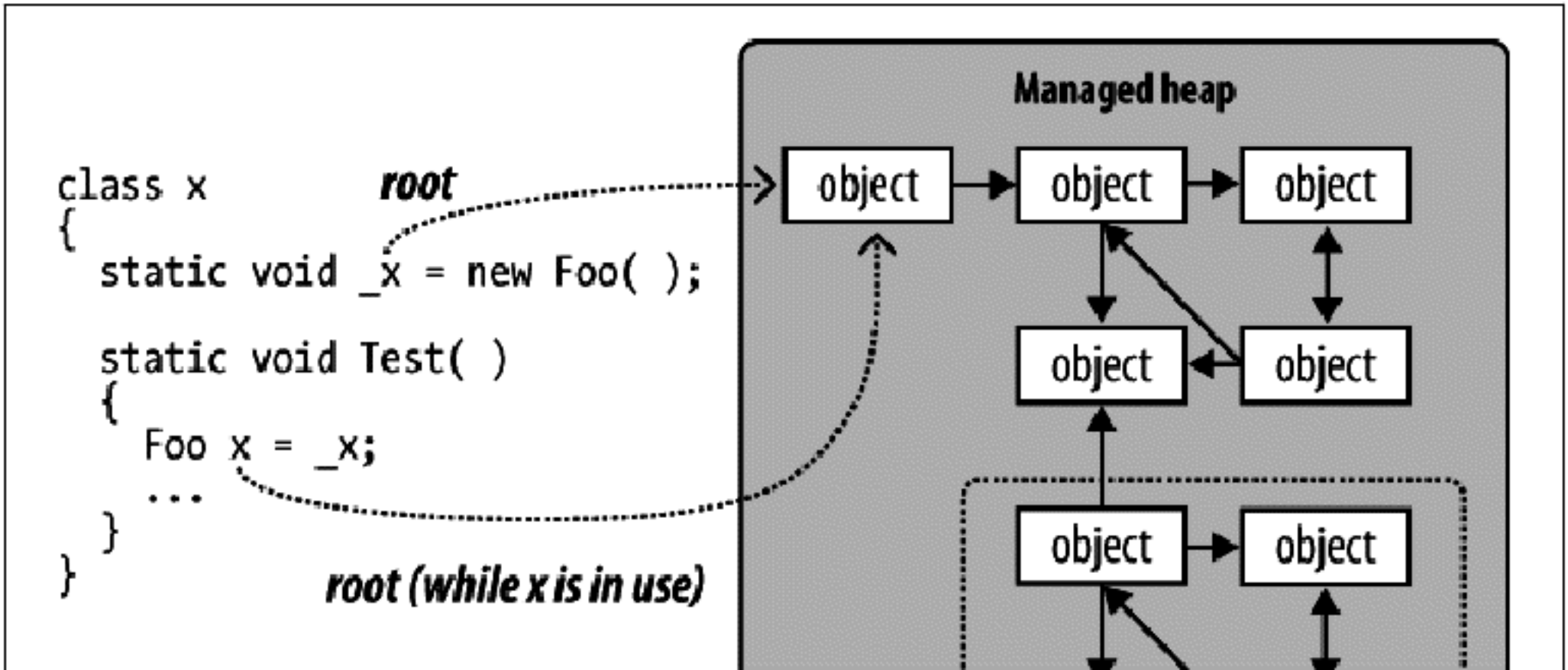
Finalizers

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

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

```
    }
  }
```

Finalizers are possible because garbage collection works in distinct phases. First, the GC identifies the unused objects ripe for deletion. Those without finalizers are



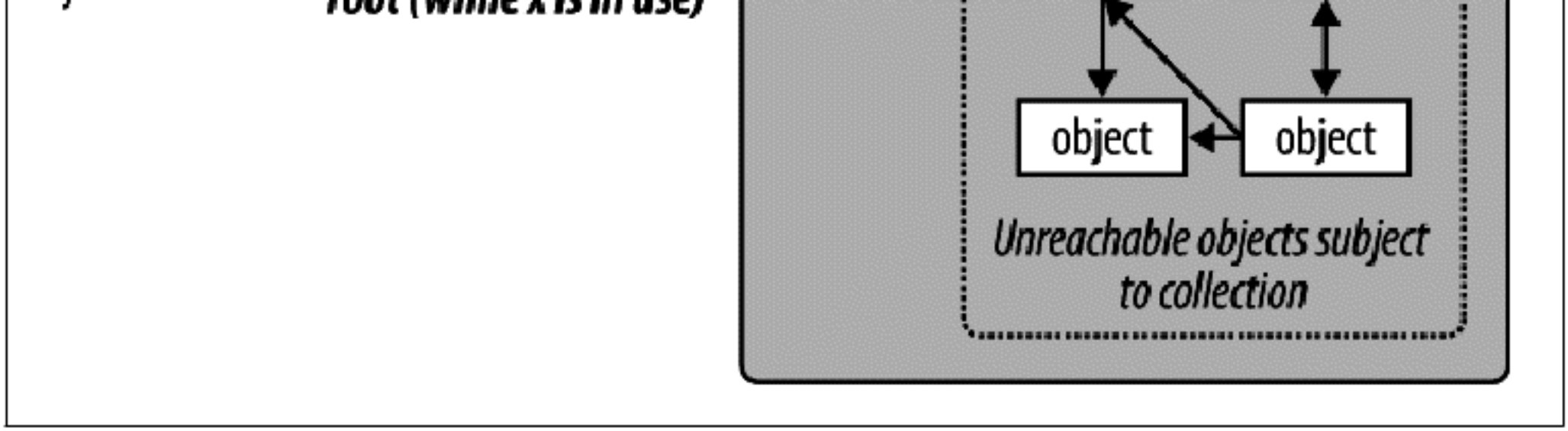


Figure 12-1. Roots

deleted right away. Those with pending (unrun) finalizers are kept alive (for now) and are put onto a special queue.

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

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

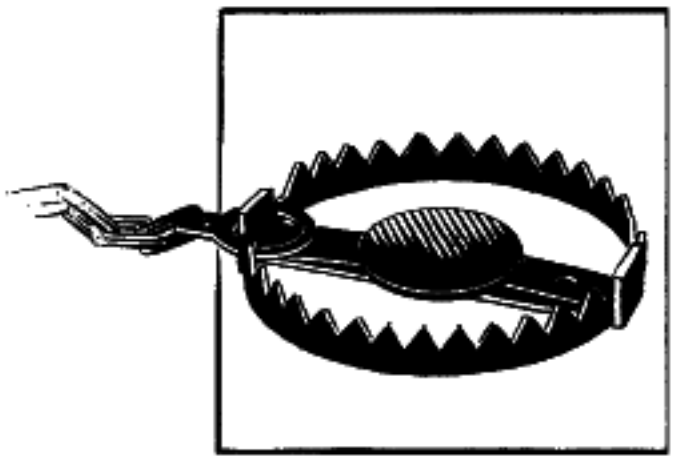
-
-
-
-
-
-

- Finalizers slow the allocation and collection of memory (the GC needs to keep track of which finalizers have run).
- Finalizers prolong the life of the object and any *referred* objects (they must all await the next garbage truck for actual deletion).
- It's impossible to predict in what order the finalizers for a set of objects will be called.
- You have limited control over when the finalizer for an object will be called.
- If code in a finalizer blocks, other objects cannot get finalized.
- Finalizers may be circumvented altogether if an application fails to unload cleanly.

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

Here are some guidelines for implementing finalizers:

- Ensure that your finalizer executes quickly.
- Never block in your finalizer (Chapter 21).
- Don't reference other finalizable objects.
- Don't throw exceptions.



An object's finalizer can get called even if an exception is thrown during construction. For this reason, it pays not to assume that

during construction. For this reason, it pays not to assume that fields are correctly initialized when writing a finalizer.

Calling Dispose from a Finalizer

One excellent use for finalizers is to provide a backup for cases when you forget to call `Dispose` on a disposable object; it's usually better to have an object disposed late than never! There's a standard pattern for implementing this, as follows:

```
class Test : IDisposable
{
  public void Dispose()
  {
    Dispose (true);
    GC.SuppressFinalize (this);
```

```
    GC.SuppressFinalize (this);
  }

  // NOT virtual
  // Prevent finalizer from running.

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

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

```
  // ...
}
```

```
    ~Test()
    {
      Dispose (false);
    }
  }
```

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

The enhanced version contains the actual disposal logic and is `protected` and `virtual`; this provides a safe point for subclasses to add their own disposal logic. The `disposing` flag means it's being called "properly" from the `Dispose` method rather than in "last-resort mode" from the finalizer. The idea is that when called with `disposing` set to `false`, this method should not, in general, reference other

with `disposing` set to `false`, this method should not, in general, reference other

objects with finalizers (because such objects may themselves have been finalized and so be in an unpredictable state). This rules out quite a lot! Here are a couple of tasks it can still perform in last-resort mode, when `disposing` is `false`:

-
-

Releasing any *direct references* to operating system resources (obtained, perhaps, via a P/Invoke call to the Win32 API)

Deleting a temporary file created on construction

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

Notice that we call `GC.SuppressFinalize` in the parameterless `Dispose` method—this prevents the finalizer from running when the GC later catches up with it. Technically, this is unnecessary, as `Dispose` methods must tolerate repeated calls. However

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

This pattern is intended more as a backup than a replacement for calling `Dispose`. A difficulty with relying on it completely is that you couple resource deallocation to memory deallocation—two things with potentially divergent interests. You also increase the burden on the finalization thread.

Resurrection

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

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

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

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

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

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

collection for other objects. We want to restrict finalization actions to those that are simple, reliable, and quick.

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

```
public class TempFileRef
{
```