15a

Isolated Storage

Each .NET program has access to a local storage area unique to that program, called *isolated storage*. Isolated storage is useful when your program can't access the standard file system, and so cannot write to `ApplicationData`, `LocalApplicationData`, `CommonApplicationData`, `MyDocuments`, and so on (see "Special Folders"). This is the case with Silverlight applications and ClickOnce applications deployed with restricted "Internet" permissions.

Isolated storage has the following disadvantages:

- The API is awkward to use.
- You can read/write only via an `IsolatedStorageStream`—you cannot obtain a file or directory path and then use ordinary file I/O.
- The machines stores (equivalent to `CommonApplicationData`) won't let users with restricted OS permissions delete or overwrite files if they were created by another user (although they can modify them). This is effectively a bug.

In terms of security, isolated storage is a fence designed more to keep you in than to keep other applications out. Data in isolated storage is strongly protected against intrusion from other .NET applications running under the most restricted permission set (i.e., the "Internet" zone). In other cases, there's no hard security preventing another application from accessing your isolated storage if it really wants to. The benefit of isolated storage is that applications must go out of their way to interfere with each other—it cannot happen through carelessness or by accident.

Applications running in a sandbox typically have their quota of isolated storage limited via permissions. The default is 1MB for Internet and Silverlight applications.

A hosted UI-based application (e.g., Silverlight) can ask the user for permission to increase the isolated storage quota by calling the `IncreaseQuotaTo` method on an `IsolatedStorageFile` object. This must be called from a user-initiated event, such as a button click. If the user agrees, the method returns true.

You can query the current allowance via the `Quota` property.

Isolation Types

Isolated storage can separate by both program and user. This results in three basic types of compartments:

Local user compartments
    One per user, per program, per computer

Roaming user compartments
    One per user, per program

Machine compartments
One per program, per computer (shared by all users of a program)

The data in a roaming user compartment follows the user across a network—with appropriate operating system and domain support. If this support is unavailable, it behaves like a local user compartment.

So far, we've talked about how isolated storage separates by "program." Isolated storage considers a program to be one of two things, depending on which mode you choose:

- An assembly
- An assembly running within the context of a particular application

The latter is called *domain isolation* and is more commonly used than *assembly isolation*. Domain isolation segregates according to two things: the currently executing assembly and the executable or web application that originally started it. Assembly isolation segregates only according to the currently executing assembly—so different applications calling the same assembly will share the same store.

Assemblies and applications are identified by their strong name. If no strong name is present, the assembly's full file path or URI is used instead. This means that if you move or rename a weakly named assembly, its isolated storage is reset.

In total, then, there are six kinds of isolated storage compartments. Table 15a-4 compares the isolation provided by each.

Table 15a-1. Isolated storage containers

Type	**Computer?**	**Application?**	**Assembly?**	**User?**	**Method to obtain store**
Domain User (default)	✓	✓	✓	✓	`GetUserStoreForDomain`
Domain Roaming		✓	✓	✓	
Domain Machine	✓	✓	✓		`GetMachineStoreForDomain`
Assembly User	✓		✓	✓	`GetUserStoreForAssembly`
Assembly Roaming			✓	✓	
Assembly Machine	✓		✓		`GetMachineStoreForAssembly`

There is no such thing as domain-only isolation. If you want to share an isolated store across all assemblies within an application, there's a simple workaround, however. Just expose a public method in one of the assemblies that instantiates and returns an `IsolatedStorageFileStream` object. Any assembly can access any isolated store if given an `IsolatedStorageFile` object—isolation restrictions are imposed upon construction, not subsequent use.

Similarly, there's no such thing as machine-only isolation. If you want to share an isolated store across a variety of applications, the workaround is to write a common assembly that all applications reference, and then expose a method on the common assembly that creates and returns an assembly-isolated `IsolatedStorageFileStream`. The common assembly must be strongly named for this to work.

Reading and Writing Isolated Storage

Isolated storage uses streams that work much like ordinary file streams. To obtain an isolated storage stream, you first specify the kind of isolation you want by calling one of the static methods on `IsolatedStorageFile`—as shown previously in Table 15a-4. You then use it to construct an `IsolatedStorageFileStream`, along with a filename and `FileMode`:

```
// IsolatedStorage classes live in System.IO.IsolatedStorage

using (IsolatedStorageFile f =
       IsolatedStorageFile.GetMachineStoreForDomain())
using (var s = new IsolatedStorageFileStream ("hi.txt",FileMode.Create,f))
using (var writer = new StreamWriter (s))
  writer.WriteLine ("Hello, World");

// Read it back:

using (IsolatedStorageFile f =
       IsolatedStorageFile.GetMachineStoreForDomain())
using (var s = new IsolatedStorageFileStream ("hi.txt", FileMode.Open, f))
using (var reader = new StreamReader (s))
  Console.WriteLine (reader.ReadToEnd());        // Hello, world
```

`IsolatedStorageFile` is poorly named in that it doesn't represent a file, but rather a *container* for files (basically, a directory).

A better (though more verbose) way to obtain an `IsolatedStorageFile` is to call `IsolatedStorageFile.GetStore`, passing in the right combination of `IsolatedStorageScope` flags (as shown in Figure 15a-6):

```
var flags = IsolatedStorageScope.Machine
          | IsolatedStorageScope.Application
          | IsolatedStorageScope.Assembly;

using (IsolatedStorageFile f = IsolatedStorageFile.GetStore (flags,
  typeof (StrongName), typeof (StrongName)))
{
  ...
```

The advantage of doing it this way is that we can tell `GetStore` what kind of *evidence* to consider when identifying our program, rather than letting it choose automatically. Most commonly, you'll want to use the strong names of your program's assemblies (as we have done in this example) because a strong name is unique and easy to keep consistent across versions.

The danger of letting the CLR choose evidence automatically is that also considers Authenticode signatures (Chapter 18). This is usually undesirable because it means that an Authenticode-related change will trigger a change of identity. In particular, if you start out without Authenticode and then later decide to add it, the CLR will see your application as different from the perspective of isolated storage, and this can mean users losing data between versions.

`IsolatedStorageScope` is a flags enum whose members you must combine in exactly the right way to get a valid store. Figure 15a-6 lists all the valid combinations. Note that they let you access the roaming stores (these are like local stores but with the capability to "roam" via Windows Roaming Profiles).

	Assembly	**Assembly & Domain**
Local User	Assembly | User	Assembly | Domain | User
Roaming User	Assembly | User | Roaming	Assembly | Domain | User | Roaming
Machine	Assembly | Machine	Assembly | Domain | Machine

Figure 15a-1. Valid IsolatedStorageScope combinations

Here’s how to write to a store isolated by assembly and roaming user:

```
var flags = IsolatedStorageScope.Assembly
          | IsolatedStorageScope.User
          | IsolatedStorageScope.Roaming;

using (IsolatedStorageFile f = IsolatedStorageFile.GetStore (flags, null, null))
using (var s = new IsolatedStorageFileStream ("a.txt", FileMode.Create, f))
using (var writer = new StreamWriter (s))
  writer.WriteLine ("Hello, World");
```

Store Location

Here’s where .NET writes isolated storage files:

Scope	Location
Local user	`[LocalApplicationData]`*\IsolatedStorage*
Roaming user	`[ApplicationData]`*\IsolatedStorage*
Machine	`[CommonApplicationData]`*\IsolatedStorage*

You can obtain the locations of each of the folders in square brackets by calling the `Environment.GetFolderPath` method. Here are the defaults for Windows Vista and above:

Scope	Location
Local user	*\Users\<user>\AppData\Local\IsolatedStorage*
Roaming user	*\Users\<user>\AppData\Roaming\IsolatedStorage*
Machine	*\ProgramData\IsolatedStorage*

For Windows XP:

Scope	Location
Local user	*\Documents and Settings\<user>\Local Settings\Application Data\IsolatedStorage*
Roaming user	*\Documents and Settings\<user>\Application Data\IsolatedStorage*
Machine	*\Documents and Settings\All Users\Application Data\IsolatedStorage*

These are merely the base folders; the data files themselves are buried deep in a labyrinth of subdirectories whose names derive from hashed assembly names. This is both a reason to use—and not to use—isolated storage. On the one hand, it makes isolation possible: a permission-restricted application wanting to interfere with another can be stumped by being denied a directory listing—despite having the same filesystem rights as its peers. On the other hand, it makes administration impractical from outside the application. Sometimes it’s handy—or essential—to edit an XML configuration file in Notepad so that an application can start up properly. Isolated storage makes this impractical.

Enumerating Isolated Storage

An `IsolatedStorageFile` object also provides methods for listing files in the store:

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForDomain())
{
  using (var s = new IsolatedStorageFileStream ("f1.x",FileMode.Create,f))
    s.WriteByte (123);

  using (var s = new IsolatedStorageFileStream ("f2.x",FileMode.Create,f))
    s.WriteByte (123);

  foreach (string s in f.GetFileNames ("*.*"))
    Console.Write (s + " ");                   // f1.x f2.x
}
```

You can also create and remove subdirectories, as well as files:

```
using (IsolatedStorageFile f = IsolatedStorageFile.GetUserStoreForDomain())
{
  f.CreateDirectory ("subfolder");

  foreach (string s in f.GetDirectoryNames ("*.*"))
    Console.WriteLine (s);                             // subfolder

  using (var s = new IsolatedStorageFileStream (@"subfolder\sub1.txt",
                                                FileMode.Create, f))
    s.WriteByte (100);

  f.DeleteFile (@"subfolder\sub1.txt");
  f.DeleteDirectory ("subfolder");
}
```

With sufficient permissions, you can also enumerate over all isolated stores created by the current user, as well as all machine stores. This function can violate program privacy, but not user privacy. Here's an example:

```
System.Collections.IEnumerator rator =
  IsolatedStorageFile.GetEnumerator (IsolatedStorageScope.User);

while (rator.MoveNext())
{
  var isf = (IsolatedStorageFile) rator.Current;

  Console.WriteLine (isf.AssemblyIdentity);      // Strong name or URI
  Console.WriteLine (isf.CurrentSize);
  Console.WriteLine (isf.Scope);                 // User + ...
}
```

The `GetEnumerator` method is unusual in accepting an argument (this makes its containing class `foreach`-unfriendly). `GetEnumerator` accepts one of three values:

`IsolatedStorageScope.User`
: Enumerates all local stores belonging to the current user

`IsolatedStorageScope.User | IsolatedStorageScope.Roaming`
: Enumerates all roaming stores belonging to the current user

`IsolatedStorageScope.Machine`
: Enumerates all machine stores on the computer

Once you have the `IsolatedStorageFile` object, you can list its content by calling `GetFiles` and `GetDirectories`.