14 Applications

APPLICATIONS HAVE SPECIAL SUPPORT in Windows Forms. For starters, you can manage and tailor your application's lifetime, and, when the work flow is disrupted by an unhandled exception, you can choose from several methods of response. Then, there are several application models that you can employ, including Single Document Interface (SDI) and Multiple Document Interface (MDI) applications, each of which can support either multiple-instance or single-instance mode, the former the VS05 default and the latter requiring special consideration. All applications, however, can discover and use a wide variety of information about the system and environment they execute in.

This chapter focuses on these topics in depth, and starts by defining what an application actually is.

Applications

An *application* is anything with an .exe extension that can be started from the Windows shell. However, applications are also provided for directly in Windows Forms by the Application class:

```
namespace System.Windows.Forms {
   sealed class Application {

      // Properties
      public static bool AllowQuit { get; }
      public static string CommonAppDataPath { get; }
      public static RegistryKey CommonAppDataRegistry { get; }
      public static string CompanyName { get; }
      public static CultureInfo CurrentCulture { get; set; }
      public static InputLanguage CurrentInputLanguage { get; set; }
      public static string ExecutablePath { get; }
      public static string LocalUserAppDataPath { get; }
      public static bool MessageLoop { get; }
```

updated

```
    public static FormCollection OpenForms { get; } // New
    public static string ProductName { get; }
    public static string ProductVersion { get; }
    public static bool RenderWithVisualStyles { get; } // New
    public static string SafeTopLevelCaptionFormat { get; set; }
    public static string StartupPath { get; }
    public static string UserAppDataPath { get; }
    public static RegistryKey UserAppDataRegistry { get; }
    public static bool UseWaitCursor { get; set; } // New
    public static VisualStyleState VisualStyleState { get; set; } // New

    // Methods
    public static void AddMessageFilter(IMessageFilter value);
    public static void DoEvents();
    public static void EnableVisualStyles();
    public static void Exit();
    public static void Exit(CancelEventArgs e); // New
    public static void ExitThread();
    public static bool FilterMessage(ref Message message); // New
    public static ApartmentState OleRequired();
    public static void OnThreadException(Exception t);
    public static void RaiseIdle(EventArgs e); // New
    public static void RegisterMessageLoop(
      MessageLoopCallback callback); // New
    public static void RemoveMessageFilter(IMessageFilter value);
    public static void Restart(); // New
    public static void Run();
    public static void Run(ApplicationContext context);
    public static void Run(Form mainForm);
    public static void SetCompatibleTextRenderingDefault(
      bool defaultValue); // New
    public static bool SetSuspendState(
      PowerState state, bool force, bool disableWakeEvent); // New
    public static void SetUnhandledExceptionMode(
      UnhandledExceptionMode mode); // New
    public static void SetUnhandledExceptionMode(
      UnhandledExceptionMode mode, bool threadScope); // New
    public static void UnregisterMessageLoop();// New

    // Events
    public static event EventHandler ApplicationExit;
    public static event EventHandler EnterThreadModal; // New
    public static event EventHandler Idle;
    public static event EventHandler LeaveThreadModal; // New
    public static event ThreadExceptionEventHandler ThreadException;
    public static event EventHandler ThreadExit;
  }
}
```

Notice that all the members of the Application class are static. Although there is per-application state in Windows Forms, there is no instance of an Application class. Instead, the Application class is a scoping mechanism for exposing the various services that the class provides, including control of application lifetime and support for message handling.

Application Lifetime

A Windows Forms application starts when the Main method is called. However, to initialize a Windows Forms application fully and start it routing Windows Forms events, you need to invoke Application.Run in one of three ways.

The first is simply to call Run with no arguments. This approach is useful only if other means have already been used to show an initial UI:

```
// Program.cs
static class Program {

  [STAThread]
  static void Main() {
    ...
    // Create and show the main form modelessly
    MainForm form = new MainForm();
    form.Show();

    // Run the application
    Application.Run();
  }
}
```

When you call Run with no arguments, the application runs until explicitly told to stop, even when all its forms are closed. This puts the burden on some part of the application to call the Application class Exit method, typically when the main application form is closing:

```
// MainForm.cs
partial class MainForm : Form {
  ...
  void MainForm_FormClosed(object sender, FormClosedEventArgs e) {
    // Close the application when the main form goes away
    // Only for use when Application.Run is called without
    // any arguments
    Application.Exit();
  }
  ...
}
```

Typically, you call Application.Run without any arguments only when the application needs a secondary UI thread. A *UI thread* is one that calls Application.Run and can process the

events that drive a Windows application. Because a vast majority of applications contain a single UI thread and because most of them have a *main form* that, when closed, causes the application to exit, another overload of the Run method is used far more often. This overload of Run takes as an argument a reference to the form designated as the main form. When Run is called in this way, it shows the main form and doesn't return until the main form closes:

```
// Program.cs
static class Program {

  [STAThread]
  static void Main() {
    ...
    // Create the main form
    MainForm form = new MainForm();

    // Run the application until the main form is closed
    Application.Run(form);
  }
}
```

In this case, there is no need for explicit code to exit the application. Instead, Application watches for the main form to close before exiting.

Application Context

Internally, the Run method creates an instance of the ApplicationContext class. ApplicationContext detects main form closure and exits the application as appropriate:

updated

```
namespace System.Windows.Forms {
  class ApplicationContext {

    // Constructors
    public ApplicationContext();
    public ApplicationContext(Form mainForm);

    // Properties
    public Form MainForm { get; set; }
    public object Tag { get; set; } // New

    // Events
    public event EventHandler ThreadExit;

    // Methods
    public void ExitThread();
    protected virtual void ExitThreadCore();
    protected virtual void OnMainFormClosed(object sender, EventArgs e);
  }
}
```

In fact, the Run method allows you to pass an ApplicationContext yourself:

```
// Program.cs
static class Program {

  [STAThread]
  static void Main() {
    ...
    // Run the application with a context
    ApplicationContext ctx = new ApplicationContext(new MainForm());
    Application.Run(ctx);
  }
}
```

This is useful if you'd like to derive from the ApplicationContext class and provide your own custom context:

```
// TimedApplicationContext.cs
class TimedApplicationContext : ApplicationContext {

  Timer timer = new Timer();

  public TimedApplicationContext(Form mainForm) : base(mainForm) {
    timer.Tick += timer_Tick;
    timer.Interval = 5000; // 5 seconds
    timer.Enabled = true;
  }

  void timer_Tick(object sender, EventArgs e) {
    timer.Enabled = false;
    timer.Dispose();

    DialogResult res =
      MessageBox.Show(
        "OK to charge your credit card?",
        "Time's Up!",
        MessageBoxButtons.YesNo);

    if( res == DialogResult.No ) {
      // See ya...
      this.MainForm.Close();
    }
  }
}

// Program.cs
static class Program {
```

```
  [STAThread]
  static void Main() {
    ...
    // Run the application with a custom context
    TimedApplicationContext ctx =
      new TimedApplicationContext(new MainForm());
    Application.Run(ctx);
  }
}
```

This custom context class waits for five seconds after an application has started and then asks to charge the user's credit card. If the answer is no, the main form of the application is closed (available from the MainForm property of the base ApplicationContext class), causing the application to exit.

You might also encounter situations when you'd like to stop the application from exiting when the main form goes away, such as an application that's serving .NET remoting clients and needs to stick around even if the user has closed the main form.[1] In these situations, you override the OnMainFormClosed method from the ApplicationContext base class:

```
// RemotingServerApplicationContext.cs
class RemotingServerApplicationContext : ApplicationContext {

  public RemotingServerApplicationContext(Form mainForm) :
    base(mainForm) {}

  protected override void OnMainFormClosed(object sender, EventArgs e) {
    // Don't let base class exit application
    if( this.IsServicingRemotingClient() ) return;

    // Let base class exit application
    base.OnMainFormClosed(sender, e);
  }

  protected bool IsServicingRemotingClient() {...}
}
```

When all the .NET remoting clients have exited, you must make sure that Application.Exit is called, in this case by calling the base ApplicationContext class's OnMainFormClosed method.

[1] .NET remoting is a technology that allows objects to talk to each other across application and machine boundaries. Remoting is beyond the scope of this book but is covered very nicely in Ingo Rammer's book *Advanced .NET Remoting* (APress, 2002).

Application Events

During the lifetime of an application, several key application events—Idle, ThreadExit, and ApplicationExit—are fired by the Application object. You can subscribe to application events at any time, but it's most common to do it in the Main function:

```
// Program.cs
static class Program {

  [STAThread]
  static void Main() {
    ...
    Application.Idle += App_Idle;
    Application.ThreadExit += App_ThreadExit;
    Application.ApplicationExit += App_ApplicationExit;

    // Run the application
    Application.Run(new MainForm());
  }

  static void App_Idle(object sender, EventArgs e) {...}
  static void App_ThreadExit(object sender, EventArgs e) {...}
  static void App_ApplicationExit(object sender, EventArgs e) {...}
}
```

The Idle event happens when a series of events have been dispatched to event handlers and no more events are waiting to be processed. The Idle event can sometimes be used to perform concurrent processing in tiny chunks, but it's much more convenient and robust to use worker threads for those kinds of activities. This technique is covered in Chapter 18: Multithreaded User Interfaces.

When a UI thread is about to exit, it receives a notification via the ThreadExit event. When the last UI thread goes away, the application's ApplicationExit event is fired.

UI Thread Exceptions

One other application-level event that is fired as necessary by the Application object is the ThreadException event. This event is fired when a UI thread causes an exception to be thrown. This one is so important that Windows Forms provides a default handler if you don't.

The typical .NET unhandled-exception behavior on a user's machine yields a dialog, as shown in Figure 14.1.[2]

[2] A developer's machine is likely to have VS05 installed, and VS05 provides a much more detailed, developer-oriented dialog.

Figure 14.1 Default .NET Unhandled-Exception Dialog

This kind of exception handling tends to make users unhappy. This dialog isn't necessarily explicit about what actually happened, even if you view the data in the error report. And worse, there is no way to continue the application to attempt to save the data being worked on at the moment. On the other hand, a Windows Forms application that experiences an unhandled exception during the processing of an event shows a more specialized default dialog like the one in Figure 14.2.

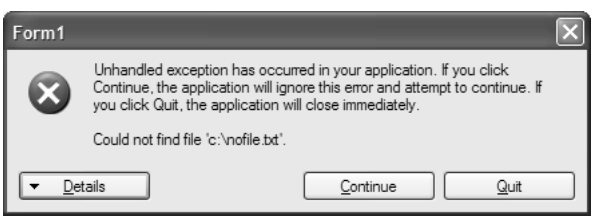

Figure 14.2 Default Windows Forms Unhandled-Exception Dialog

This dialog is the ThreadExceptionDialog (from the System.Windows.Forms namespace), and it looks functionally the same as the one in Figure 14.1, with one important difference: The Windows Forms version has a Continue button. What's happening is that Windows Forms itself catches exceptions thrown by event handlers; in this way, even if that event handler caused an exception—for example, if a file couldn't be opened or there was a security violation—the user is allowed to continue running the application with the hope that saving will work, even if nothing else does. This safety net makes Windows Forms applications more robust in the face of even unhandled exceptions than Windows applications of old.

However, if an unhandled exception is caught, the application could be in an inconsistent state, so it's best to encourage your users to save their files and restart the application. To implement this, you replace the Windows Forms unhandled-exception dialog with an application-specific dialog by handling the application's thread exception event:

```
// Program.cs
static class Program {

  [STAThread]
  static void Main() {
    // Handle unhandled thread exceptions
    Application.ThreadException += App_ThreadException;
    ...
    // Run the application
    Application.Run(new MainForm());
  }

  static void App_ThreadException(
    object sender, ThreadExceptionEventArgs e) {
    // Does user want to save or quit?
    string msg =
      "A problem has occurred in this application:\r\n\r\n" +
      "\t" + e.Exception.Message + "\r\n\r\n" +
      "Would you like to continue the application so that\r\n" +
      "you can save your work?";
    DialogResult res = MessageBox.Show(
      msg,
      "Unexpected Error",
      MessageBoxButtons.YesNo);
      ...
  }
}
```

Notice that the thread exception handler takes a ThreadExceptionEventArgs object, which includes the exception that was thrown. This is handy if you want to tell the user what happened, as shown in Figure 14.3.

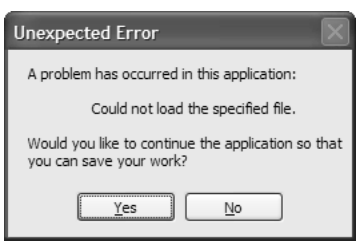

Figure 14.3 Custom Unhandled-Exception Dialog

If the user wants to return to the application to save work, all you need to do is return from the ThreadException event handler. If, on the other hand, the user decides not to continue with the application, calling Application.Exit shuts down the application. Both are shown here:

```
// Program.cs
static class Program {
  ...
  static void App_ThreadException(
    object sender, ThreadExceptionEventArgs e) {
    ...
    // Save or quit
    DialogResult res = MessageBox.Show(...);

    // If save: returning to continue the application and allow saving
    if( res == DialogResult.Yes ) return;

    // If quit: shut 'er down, Clancy, she's a'pumpin' mud!
    Application.Exit();
}
```

Handling exceptions in this way gives users a way to make decisions about how an application will shut down, if at all, in the event of an exception. However, if it doesn't make sense for users to be involved in unhandled exceptions, you can make sure that the ThreadException event is never fired. Call Application.SetUnhandledExceptionMode:

```
Application.SetUnhandledExceptionMode(
  UnhandledExceptionMode.ThrowException);
```

Although it's not obvious from the enumeration value's name, this code actually prevents ThreadException from being fired. Instead, it dumps the user straight out of the application before displaying the .NET unhandled-exception dialog from Figure 14.1:

```
namespace System.Windows.Forms {
  enum UnhandledExceptionMode {
    Automatic = 0, // default
    ThrowException = 1, // Never fire Application.ThreadException
    CatchException = 2, // Always fire Application.ThreadException
  }
}
```

In general, the behavior exhibited by UnhandledExceptionMode.ThrowException isn't the most user friendly, or informative, when something catastrophic happens. Instead, it's much better to involve users in deciding how an application shuts down.

Going the other way, you can also use command line arguments to let users make decisions about how they want their application to start up.

Passing Command Line Arguments

Command line arguments allow users to determine an application's initial state and operational behavior when launched.[3] Before command line arguments can be processed to express a user's wishes, they need to be accessed. To do this, you change your application's entry point method, Main, to accept a string array to contain all the passed arguments:

```
// Program.cs
static class Program {
  [STAThread]
  static void Main(string[] args) {
    ...
  }
}
```

.NET constructs the string array by parsing the command line string, which means extracting substrings, delimited by spaces, and placing each substring into an element of the array. Command line syntax, which dictates which command line arguments your application can process and the format they should be entered in, is left up to you. Here is one simple approach:

```
// Program.cs
static class Program {
  [STAThread]
  static void Main(string[] args) {
    ...
    bool flag = false;
    string name = "";
    int number = 0;

    // *Very* simple command line parsing
    for( int i = 0; i != args.Length; ++i ) {
      switch( args[i] ) {
        case "/flag": flag = true; break;
        case "/name": name = args[++i]; break;
        case "/number": number = int.Parse(args[++i]); break;
        default: MessageBox.Show("Invalid args!"); return;
      }
    }
    ...
  }
}
```

[3] Application and user settings are another mechanism for doing so, and they are covered in Chapter 15: Settings.

If your static Main method isn't where you want to handle the command line arguments for your application session, GetCommandLineArgs can come in handy for retrieving the command line arguments for the current application session:[4]

```
// Program.cs
static class Program {
  [STAThread]
  static void Main() {
    ...
    string[] args = Environment.GetCommandLineArgs();

    // *Very* simple command line parsing
    // Note: Starting at item [1] because args item [0] is exe path
    for( int i = 1; i != args.Length; ++i ) {
      ...
    }
    ...
  }
}
```

You can see that GetCommandLineArgs always returns a string array with at least one item: the executable path.

Processing command line arguments is relatively straightforward, although special types of applications, known as single-instance applications, need to process command line arguments in special ways.

Single-Instance Applications

By default, each EXE is an application that has an independent lifetime, even if multiple instances of the same application are running at the same time. However, it's common to want to limit an EXE to a single instance, whether it's an SDI application with a single top-level window, an MDI application, or an SDI application with multiple top-level windows. All these kinds of applications require that another instance detect the initial instance and then cut its own lifetime short.

Single-Instance Detection and Management

new

You could build a custom single-instance application using custom code that incorporates threading and .NET remoting. However, the VB.NET runtime library, Microsoft.VisualBasic.dll, contains a class that provides such an implementation for you: WindowsFormsApplicationBase, located in the Microsoft.VisualBasic.ApplicationServices

[4] If you want to see more robust command line parsing support, see the Genghis class library, which is available at http://www.genghisgroup.com (http://tinysells.com/8).

namespace.[5] WindowsFormsApplicationBase does not inherit from the Application class in System.Windows.Forms, but WindowsFormsApplicationBase is designed to replace the use of the Application class to run and manage an application's lifetime, as you'll see shortly.

If you are using C#, you add a reference to this assembly by right-clicking the project and selecting Add Reference from the context menu. From the .NET tab of the subsequently loaded Add Reference dialog, select Microsoft.VisualBasic.dll. When this DLL is referenced, you derive from WindowsFormsApplicationBase before extending your custom class with support for single-instance applications and passing command line arguments:

```
// SingleInstanceApplication.cs
using Microsoft.VisualBasic.ApplicationServices;
...
class SingleInstanceApplication : WindowsFormsApplicationBase {...}
```

Next, you configure SingleInstanceApplication to support single-instance applications. Set the SingleInstanceApplication class's IsSingleInstance property (implemented by the base WindowsFormsApplicationBase class) to true:

```
// SingleInstanceApplication.cs
class SingleInstanceApplication : WindowsFormsApplicationBase {

  // Must call base constructor to ensure correct initial
  // WindowsFormsApplicationBase configuration
  public SingleInstanceApplication() {

    // This ensures the underlying single-SDI framework is employed,
    // and OnStartupNextInstance is fired
    this.IsSingleInstance = true;
  }
}
```

IsSingleInstance is false by default, and the constructor is a great place to change this situation. To incorporate this into your application, replace the standard application start-up logic from your application's entry point. Then, use the following code to create an instance of your custom WindowsFormsApplicationBase type:

```
// Program.cs
static class Program {
```

[5] It's difficult to determine why this nice feature wasn't folded into the .NET Framework, which would explicitly expose it to all languages. However, Microsoft.VisualBasic.dll ships with the .NET Framework, so it's available to any .NET language, in spite of its name.

```
  [STAThread]
  static void Main(string[] args) {
    Application.EnableVisualStyles();
    SingleInstanceApplication application =
      new SingleInstanceApplication();
    application.Run(args);
  }
}
```

WindowsFormsApplicationBase exposes the Run method—the Application.Run method analog—which you invoke to open the main application form. Additionally, WindowsFormsApplicationBase.Run expects a string array containing command line arguments; passing null causes an exception to be thrown.

To specify which form is the main application form, you override WindowsFormsApplicationBase.OnCreateMainForm and set WindowsFormsApplicationBase.MainForm appropriately:

```
// SingleInstanceApplication.cs
class SingleInstanceApplication : WindowsFormsApplicationBase {
  ...
  protected override void OnCreateMainForm() {
    this.MainForm = new MainForm();
  }
}
```

As a final flourish, you can expose your custom WindowsFormsApplicationBase type via a static instantiation-helper method and thereby cut down on client code:

```
// SingleInstanceApplication.cs
class SingleInstanceApplication : WindowsFormsApplicationBase {

  static SingleInstanceApplication application;
  internal static SingleInstanceApplication Application {
    get {
      if( application == null ) {
        application = new SingleInstanceApplication();
      }
      return application;
    }
  }
  ...
}

// Program.cs
static class Program {
  ...
```

```
    [STAThread]
    static void Main(string[] args) {
      Application.EnableVisualStyles();
      SingleInstanceApplication.Application.Run(args);
    }
}
```

The effect of SingleInstanceApplication is to restrict an application to only one instance, no matter how many times it is executed. This single-instance scheme works fine as is, but it works better when the first instance of the application has a need to get command line arguments from any subsequent instances. Multiple-SDI and single-MDI applications are examples of applications that use this kind of processing.

Multiple-SDI Applications

A *multiple-SDI* application has multiple windows for content, although each window is a top-level window. Internet Explorer and Office 2003 are popular examples of multiple-SDI applications.[6] Figure 14.4 shows an example of a multiple-SDI application.

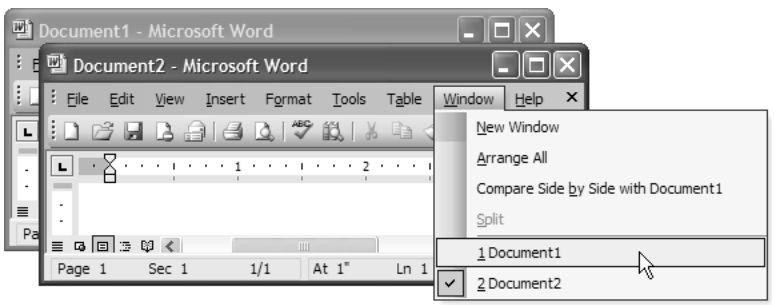

Figure 14.4 A Sample Multiple-SDI Application

A multiple-SDI application typically has the following features:

- A single instance of the application is running.
- Multiple top-level windows are running independently of each other.
- It doesn't reopen files that are currently loaded.
- When the last window goes away, the application does, too.
- A Window menu allows a user to see and select from the currently available windows.

[6] Internet Explorer can be configured to show each top-level window in its own process, making it an SDI application, or to share all windows in a single process, making it a multiple-SDI application.

When a document is created or opened, it is loaded into a new window each time, whether the file was requested via the menu system or the command line. The first time the application is called, the first new instance of the top-level form is created and set as the main application form instance; if a file was requested, it is also opened by the form.

Subsequent requests to the application are routed to the custom WindowsFormsApplicationBase object located in the already-running application instance. Each request is handled to create a new form and build up the appropriate menu structures to support navigation between top-level instances, as well as opening and closing existing top-level instances. Figure 14.5 illustrates the work flow.

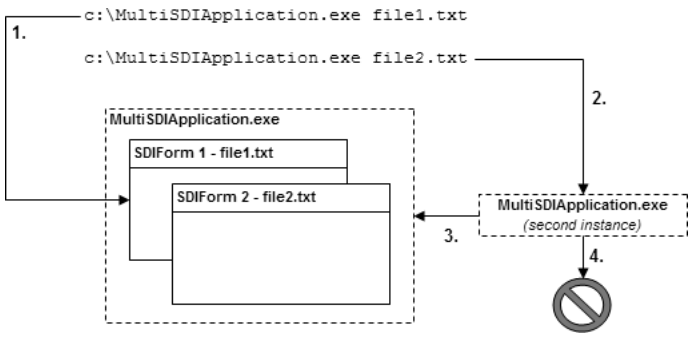

Figure 14.5 Work Flow of a Multiple-SDI Application with Support for Command Line Argument Passing

Multiple SDI requires single-instance support, which we acquire by deriving from WindowsFormsApplicationBase, as you saw earlier. We also need to ensure that the application stops running only after all top-level forms have been closed. We make the appropriate configurations from the constructor of the custom WindowsFormsApplicationBase class:

```
// MultiSDIApplication.cs
class MultiSDIApplication : WindowsFormsApplicationBase {

  static MultiSDIApplication application;
  internal static MultiSDIApplication Application {
    get {
      if( application == null ) {
        application = new MultiSDIApplication();
      }
      return application;
    }
  }
  public MultiSDIApplication() {
```

```
      // This ensures the underlying single-SDI framework is employed,
      // and OnStartupNextInstance is fired
      this.IsSingleInstance = true;

      // Needed for multiple SDI because no form is the main form
      this.ShutdownStyle = ShutdownMode.AfterAllFormsClose;
    }
  }
```

By default, the ShutdownStyle for a WindowsFormsApplicationBase object is AfterMainFormCloses, which refers to the form specified as the main form. However, with a multiple-instance SDI application, no form is the main form; therefore, no matter which form was created first, we want the application to close only after the last remaining top-level form is closed, and hence the need to explicitly set ShutdownStyle to AfterAllFormsClose.

Next, MultiSDIApplication must handle the first execution of the application. It does this by overriding OnCreateMainForm to create a new TopLevelForm object:

```
// MultiSDIApplication.cs
class MultiSDIApplication : WindowsFormsApplicationBase {
  ...
  public MultiSDIApplication() {...}

  // Create first top-level form
  protected override void OnCreateMainForm() {
    this.MainForm = this.CreateTopLevelWindow(this.CommandLineArgs);
  }

  TopLevelForm CreateTopLevelWindow(
    ReadOnlyCollection<string> args) {
    // Get file name, if provided
    string fileName = (args.Count > 0 ? args[0] : null);

    // Create a new top-level form
    return TopLevelForm.CreateTopLevelWindow(fileName);
  }
}
```

In this code, if a file argument was passed, a request is made to the main form to open it. Because all forms in a multiple-instance SDI application are top-level, however, no form is actually the main form. However, we must specify one if we override OnCreateMainForm, which helps later when the application needs to know which of the top-level forms is the active form. OnCreateMainForm passes the command line args—supplied by WindowsFormsApplicationBase.CommandLineArgs—to the helper Create TopLevelWindow method, which parses the args for a file name, passing whatever it finds to the static CreateTopLevelWindow method that's implemented by TopLevelForm. CreateTopLevel Window is static because no specific form instance is responsible for creating another form.

To cope with subsequent requests to launch the application, we again override OnStartup NextInstance:

```
// MultiSDIApplication.cs
class MultiSDIApplication : WindowsFormsApplicationBase {
  ...
  public MultiSDIApplication() {...}

  // Create first top-level form
  protected override void OnCreateMainForm() {...}

  // Create subsequent top-level form
  protected override void OnStartupNextInstance(
    StartupNextInstanceEventArgs e) {
    this.CreateTopLevelWindow(e.CommandLine);
  }

  TopLevelForm CreateTopLevelWindow(
    ReadOnlyCollection<string> args) {...}
}
```

Here, the helper CreateTopLevelWindow is again passed command line arguments and called upon to create a new top-level window, opening a file if necessary.

Multiple-instance SDI applications also allow files to be opened from existing top-level forms via the File | Open menu, something we implement using the same static Create TopLevelWindow method to open files from the command line:

```
// TopLevelForm.cs
partial class TopLevelForm : Form {
  ...
  string fileName;
  ...
  public static TopLevelForm CreateTopLevelWindow(string fileName) {
    // Detect whether file is already open
    if( !string.IsNullOrEmpty(fileName) ) {
      foreach( TopLevelForm openForm in Application.OpenForms ) {
        if( string.Compare(openForm.FileName, fileName, true) == 0 ) {
          // Bring form to top
          openForm.Activate();
          return openForm;
        }
      }
    }

    // Create new top-level form and open file
    TopLevelForm form = new TopLevelForm();
    form.OpenFile(fileName);
    form.Show();
```

```
    // Bring form to top
    openForm.Activate();
    return form;
  }

  void openToolStripMenuItem_Click(object sender, EventArgs e) {
    // Open new window
    if( this.openFileDialog.ShowDialog() == DialogResult.OK ) {
      TopLevelForm.CreateTopLevelWindow(this.openFileDialog.FileName);
    }
  }
  ...
  void OpenFile(string fileName) {
    this.fileName = fileName;
    using( StreamReader reader = new StreamReader(fileName) ) {
      textBox.Text = reader.ReadToEnd();
    }
    this.Text = this.Text + " (" + this.fileName + ")";
  }

  string FileName {
    get { return this.fileName; }
  }
}
```

CreateTopLevelWindow contains the code to check whether the desired file is already opened and, if it is, to bring the top-level window that contains it to the foreground; otherwise, the file is opened into a new top-level window.

Multiple-instance SDI applications also typically allow the creation of new files from the command line or from the File | New Window menu of a currently open top-level form. We tweak the OpenFile method to not open a file if null or if an empty string was passed as the file name:

```
// TopLevelForm.cs
partial class TopLevelForm : Form {
  ...
  static int formCount = 0;

  public TopLevelForm() {
    InitializeComponent();

    // Set form count
    ++formCount;
    this.Text += ": " + formCount.ToString();
  }
  ...
  public static TopLevelForm CreateTopLevelWindow(string fileName) {
    ...
```

```
    // Create new top-level form and open file
    TopLevelForm form = new TopLevelForm();
    form.OpenFile(fileName);
    form.Show();
    ...
  }

  void newWindowToolStripMenuItem_Click(
    object sender, EventArgs e) {
    // Open new window
    TopLevelForm.CreateTopLevelWindow(null);
  }
  ...
  void OpenFile(string fileName) {
    this.fileName = fileName;
    if( !string.IsNullOrEmpty(fileName) ) {
      using( StreamReader reader = new StreamReader(fileName) ) {
        textBox.Text = reader.ReadToEnd();
      }
    }
    else this.fileName = "Untitled" + formCount.ToString();
    this.Text = this.Text + " (" + this.fileName + ")";
  }
  ...
}
```

Because a new file doesn't have a name, the top-level form gives it one; the standard naming convention for a new file is the concatenation of some default text with a version number. In this example, we use a combination of "Untitled" and an incremental count of the number of opened top-level forms, for uniqueness.

As mentioned before, a multiple-SDI application should implement a menu that allows users to navigate between open top-level forms as this is easier when files have unique names. MultiSDIApplication is an appropriate location for this logic because it manages the application:

```
// MultiSDIApplication.cs
class MultiSDIApplication : WindowsFormsApplicationBase {
  ...
  public void AddTopLevelForm(Form form) {

    // Add form to collection of forms and
    // watch for it to activate and close
    form.Activated += Form_Activated;
    form.FormClosed += Form_FormClosed;
```

```
    // Set initial top-level form to activate
    if( this.OpenForms.Count == 1 ) this.MainForm = form;
  }

  void Form_Activated(object sender, EventArgs e) {
    // Set the currently active form
    this.MainForm = (Form)sender;
  }

  void Form_ FormClosed(object sender, FormClosedEventArgs e) {
    // Set a new "main" if necessary
    if( ((Form)sender == this.MainForm) &&
        (this.OpenForms.Count > 0) ) {
      this.MainForm = (Form)this.OpenForms[0];
    }

    form.Activated -= Form_Activated;
    form.FormClosed -= Form_FormClosed;
  }
}
```

The MultiSDIApplication class uses the AddTopLevelForm method to keep track of a list of top-level forms as they are added. Each new form is kept in a collection and is watched for Activated and FormClosed events. When a top-level form is activated, it becomes the new "main" form, which is the one whose closure is detected by the base ApplicationContext class. When a top-level form closes, it's removed from the list. If the closed form was the main form, another form is promoted to that lofty position. When the last form goes away, the base ApplicationContext class notices and exits the application.

To keep the context up-to-date with the current list of top-level forms, the custom context watches for the Closed event on all forms. In addition, the custom context needs to be notified when a new top-level form has come into existence, a task that is best handled by the new form itself:

```
// TopLevelForm.cs
partial class TopLevelForm : Form {
  ...
  public TopLevelForm() {
    ...
    // Add new top-level form to the application context
    MultiSDIApplication.Application.AddTopLevelForm(this);
    ...
  }
  ...
}
```

The only remaining task is to designate and populate the Window menu with one menu item for each top-level form. The forms themselves can do this by handling the DropDownOpening event on the ToolStripMenuItem's Window object, using that opportunity to build the list of submenu items based on the names of all the forms. However, this code is boilerplate, so it's a good candidate to be handled by MultiSDIApplication on behalf of all top-level windows, from the AddWindowMenu method:

```
// MultiSDIApplication.cs
class MultiSDIApplication : WindowsFormsApplicationBase {
  ...
  public void AddWindowMenu(ToolStripMenuItem windowMenu) {
    // Handle tool strip menu item's drop-down opening event
    windowMenu.DropDownOpening += windowMenu_DropDownOpening;
  }
}
```

Each top-level form with a Window menu can add it to the context, along with itself, when it's created:

```
// TopLevelForm.cs
partial class TopLevelForm : Form {
  ...
  public TopLevelForm() {
    ...
    // Add Window ToolStripMenuItem to the application context
    MultiSDIApplication.Application.AddWindowMenu(
      this.windowToolStripMenuItem);
    ...
  }
  ...
}
```

Now, when the Window menu is shown on any top-level window, the DropDownOpening event fires. This constructs a new menu showing the currently open top-level forms during the time gap between mouse click and menu display:

```
// MultiSDIApplication.cs
class MultiSDIApplication : WindowsFormsApplicationBase {
  ...
  void windowMenu_DropDownOpening(object sender, EventArgs e) {
    ToolStripMenuItem menu = (ToolStripMenuItem)sender;

    // Clear current menu
    if( menu.DropDownItems.Count > 0 ) {
      menu.DropDown.Dispose();
    }
    menu.DropDown = new ToolStripDropDown();
```

```
      // Populate menu with one item for each open top-level form
      foreach( Form form in this.OpenForms ) {
        ToolStripMenuItem item = new ToolStripMenuItem();
        item.Text = form.Text;
        item.Tag = form;
        menu.DropDownItems.Add(item);
        item.Click += WindowMenuItem_Click;

        // Check menu item that represents currently active window
        if( form == this.MainForm ) item.Checked = true;
      }
    }
  }
```

As each menu item is added to the Window menu, a handler is added to the Click event so that the appropriate form can be activated when it's selected. The form associated with the ToolStripMenuItem's Tag property is extracted and activated:

```
// MultiSDIApplication.cs
class MultiSDIApplication : WindowsFormsApplicationBase {
  ...
  void WindowMenuItem_Click(object sender, EventArgs e) {
    // Activate top-level form based on selection
    ((Form)((ToolStripMenuItem)sender).Tag).Activate();
  }
  ...
}
```

That's it. The extensible lifetime management of Windows Forms applications via a custom application context, along with a helper to find and activate application instances already running, provides all the help we need to build a multiple-SDI application in only a few lines of code. The result is shown in Figure 14.6.

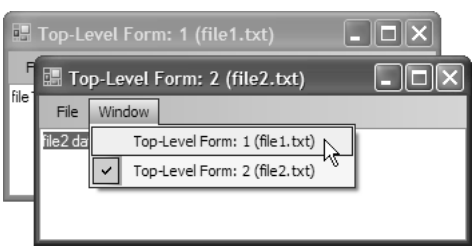

Figure 14.6 Multiple-Instance SDI Application in Action

Multiple-SDI applications share much in common with MDI applications, although each document in an MDI application is loaded into a child window rather than a new main window. The key similarities include the requirement for MDI applications to be managed from a single executable and the ability to handle command line parameters.

Single-MDI Applications

Consider an MDI application like Microsoft Excel; files opened from the file system (by double-clicking) are all opened as separate child windows within the parent Excel window.[7] For the first instance of an MDI application to open a new child window to display the file that was passed to the second instance of the application, the second instance must be able to communicate with the initial instance.

A single-MDI application exhibits the characteristics we described in Chapter 2: Forms, as well as the following features:

- A single instance of the application is running.
- Multiple MDI child windows are running within the same MDI parent window.
- Currently opened files are not reopened.
- When the last MDI child window goes away, the application remains.
- When the MDI parent window goes away, the application exits.
- A Window menu allows a user to see and select from the currently available windows.

The work flow for a single-MDI application ensures that a new MDI child form is opened each time the application is called, whether or not a file was requested for opening.

The first time the application is called, the MDI parent is created and set as the main application form instance; if a file was requested, it is also opened into a new MDI child form. Subsequent requests to the application are routed through the MDI parent form to create a new MDI child form and build up the appropriate menu structures to support navigation between top-level instances, as well as opening and closing existing top-level instances. Figure 14.7 illustrates the work flow.

[7] The fundamentals of building an MDI application in Windows Forms are described in Chapter 2: Forms.

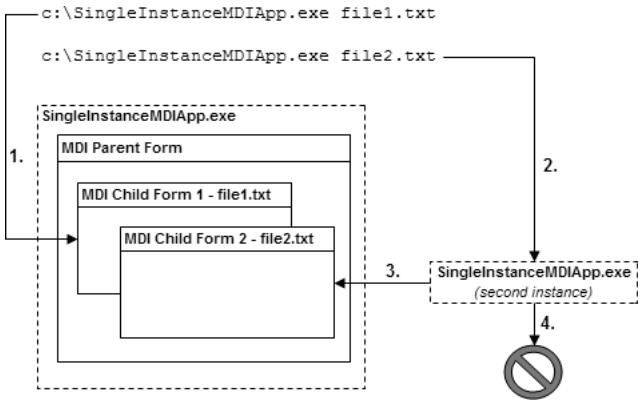

Figure 14.7 Work Flow of a Single-MDI Application with Support for Passing Command Line Arguments

With WindowsFormsApplicationBase ensuring that only one instance of the application executes, we need to handle two specific scenarios: first, when arguments are passed from the command line directly when the first instance loads and, second, when the first instance is passed command line arguments from a second instance.

Handling the first scenario requires a main application form that's an MDI parent and can open a new or existing file into an MDI child form:

```
// MDIParentForm.cs
partial class MDIParentForm : Form {
  ...
  // This is necessary to bring the MDI parent window to the front,
  // because Activate and BringToFront don't seem to have any effect.
  [DllImport("user32.dll")]
  static extern bool SetForegroundWindow(IntPtr hWnd);

  public void CreateMDIChildWindow(string fileName) {

    SetForegroundWindow(this.Handle);

    // Detect whether file is already open
    if( !string.IsNullOrEmpty(fileName) ) {
      foreach( MDIChildForm openForm in this.MdiChildren ) {
        if( string.Compare(openForm.FileName, fileName, true) == 0 ) {
          openForm.Activate();
          return;
        }
      }
    }
```

```
    // If file not open, open it
    MDIChildForm form = new MDIChildForm();
    form.OpenFile(fileName);
    form.MdiParent = this;
    form.Show();
  }

  void newToolStripMenuItem_Click(object sender, EventArgs e) {
    this.CreateMDIChildWindow(null);
  }

  void openToolStripMenuItem_Click(object sender, EventArgs e) {
    if( this.openFileDialog.ShowDialog() == DialogResult.OK ) {
      this.CreateMDIChildWindow(this.openFileDialog.FileName);
    }
  }
  ...
}
```

This code allows users to open a file using a menu strip item, and it lays the foundation for opening a file from the command line, including preventing the reopening of an already open file. We continue using WindowsFormsApplicationBase to achieve this, updating the earlier sample to acquire the command line arguments and pass them to the application main form's CreateMDIChildWindow method to open a file:

```
// SingleMDIApplication.cs
class SingleMDIApplication : WindowsFormsApplicationBase {

  static SingleMDIApplication application;
  internal static SingleMDIApplication Application {
    get {
      if( application == null ) {
        application = new SingleMDIApplication();
      }
      return application;
    }
  }

  public SingleMDIApplication() {
    // This ensures the underlying single-SDI framework is employed,
    // and OnStartupNextInstance is fired
    this.IsSingleInstance = true;
  }

  // Load MDI parent form and first MDI child form
  protected override void OnCreateMainForm() {
```

```
      this.MainForm = new MDIParentForm();
      this.CreateMDIChildWindow(this.CommandLineArgs);
    }

    void CreateMDIChildWindow(ReadOnlyCollection<string> args) {
      // Get file name, if provided
      string fileName = (args.Count > 0 ? args[0] : null);

      // Ask MDI parent to create a new MDI child
      // and open file
      ((MDIParentForm)this.MainForm).CreateMDIChildWindow(fileName);
    }
  }
```

During construction, we specify that this application is a single-instance application. Unlike with multiple-SDI applications, however, we don't need to set the ShutdownStyle property because its value defaults to AfterMainFormCloses—exactly what is needed for an MDI application.

OnCreateMainForm creates the MDI parent form and sets it as the application's main form and the one responsible for creating MDI child windows. Then, the command line arguments are passed to the helper CreateMDIChildWindow method, which parses them for a file name. Either a file name or null is passed to the MDI parent form's version of CreateMDIChildWindow, which creates the new MDI child window, into which it loads a file; then CreateMDIChildWindow establishes the MDI parent-child relationship and shows the requested file. CreateMDIChildWindow also activates the MDI parent form to bring the application to the foreground.

In the second scenario, the desired processing is for the command line arguments to be passed from the second instance to the first, to which the first instance responds by processing the command line arguments and, if required, creating a new MDI child form. WindowsFormsApplicationBase handles the underlying mechanics of passing arguments from the second instance to the first, but it is up to you to process the command line arguments accordingly. You can achieve this by overriding WindowsFormsApplicationBase.OnStartupNextInstance, which passes the command line arguments via the CommandLine property of a StartupNextInstanceEventArgs object. The following code shows the OnStartupNextInstance override implementation:

```
// SingleMDIApplication.cs
class SingleMDIApplication : WindowsFormsApplicationBase {
  ...
  // Must call base constructor to ensure correct initial
  // WindowsFormsApplicationBase configuration
  public SingleMDIApplication() {...}
```

```
    // Load MDI parent form and first MDI child form
    protected override void OnCreateMainForm() {...}

    // Load subsequent MDI child form
    protected override void OnStartupNextInstance(
      StartupNextInstanceEventArgs e) {
      this.CreateMDIChildWindow (e.CommandLine);
    }

    void CreateMDIChildWindow(ReadOnlyCollection<string> args) {...}
}
```

As you can see, centralizing CreateMDIChildWindow into a single helper method greatly simplifies the implementation of OnStartupNextInstance.

That's the complete solution, so let's look at how it operates. Suppose we start the application for the first time by executing the following statement from the command line:

```
C:\SingleInstanceSample.exe C:\file1.txt
```

The result is to load the application, configure the single-instance command line argument (passing support from our derivation of WindowsFormsApplicationBase), load the main MDI parent form, and, finally, open an MDI child form, displaying the file specified from the command line arguments. Figure 14.8 illustrates the result.

Figure 14.8 Result of Creating a First Instance of a Single-Instance Application

Now, consider the next statement being called while the first instance is still executing:

```
C:\SingleInstanceSample.exe C:\file2.txt
```

This time, a second instance of the application is created, but—thanks to SingleMDIApplication, our WindowsFormsApplicationBase derivation—the second instance passes its command line arguments to the first instance before closing itself down. The first instance processes the incoming command line arguments from OnStartupNextInstance,

requesting the MDI parent form to open a new MDI child and display the specified file. The result is shown in Figure 14.9.

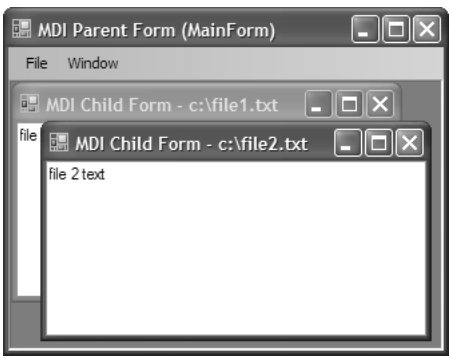

Figure 14.9 Result of Creating a Second Instance of a Single-Instance Application

Although it would be difficult to code single-instance applications such as single MDI and multiple SDI by hand, the presence of support in the Visual Basic runtime assembly makes life a lot easier. This is one of the strengths of Windows Forms; unlike forms packages of old, Windows Forms is only one part of a much larger, integrated whole. When its windowing classes don't meet your needs, you still have all the rest of the .NET Framework Class Library to fall back on.

Where Are We?

The seemingly simple application architecture in Windows Forms and .NET provides some useful capabilities, including tailored lifetime support and support for building SDI and MDI applications, whether multiple or single-instance.