Using Multiple Forms, Scope and Lifetime, Assembly Properties

A VB.net Windows (GUI) project can be composed of many windows (or forms). This can be much more "polished" (professional looking) than using MessageBox.Show.

The initial form window is called the ***startup form***. To add additional forms, just use the menu Project-->Add Windows Form..., and select one of the ***template*** forms. (A template form is a pre-build window with controls already added.) The plain form ("Windows Form") is the default template selected.

One thing to do right away is to rename your form from "Form2.vb" to something useful. Note the name is really the name of the source code file created to support the form, "SummaryForm.vb" for example.

Most Windows applications have a Help menu item called "About *nameOfApplication*" that displays all sort of information about the application: official name, version number, patching version, plug-in information, copyright, URL of company, and sometimes other information (author(s), description of application, links for on-line support, updates, or add-ons, ...) You should add such a form for all professional projects.

> A notable exception is newer Windows applications that use a Ribbon interface that don't have a menu bar at all. Instead, click on the clover-leaf button and choose "*nameOfApplication* Options". Then click on the "Resources" button an you will see the "About" button.

Using the Solution Explorer Window

Once you have more than one form you will find yourself using the solution explorer window more often. This shows all the files included in the current project. Using this window you can add files to and remove files from the project, view code or the Designer view, and view the properties of a form or the whole project. (You can also do these tasks using the Project menu.)

Setting Assembly Properties

Recall that in VB.net, one or more projects can be part of a single *assembly*. (Most of the time there will only be one project in an assembly.) "Assembly" is the .net name for an executable file (application or DLL). Windows access controls are based on assemblies, which can be digitally signed. One of the things the CLR needs to know is the version number of components (other assemblies) used by the assembly; if not found your assembly won't run. See Understanding And Using Assemblies and Namespaces in .NET.

Assemblies can use different versions of the core .NET component libraries (which are assemblies too) such as mscorlib, Microsoft.VisualBasic, and System. Every component used by an assembly has a version number to identify it. This is why you can run several versions of .net on the same host; each application knows the versions of components that

it needs. Assemblies can be verified by passing an encryption key to prevent unauthorized changes.

You can set some properties for the assembly. (And some of these properties can be viewed by right-clicking on a `.exe` or `.dll` file and selecting *properties*.) Some of the information you can set includes the application's name, description, and version number. There are in fact many properties that you can set, but the GUI dialog in VS only shows a few; you would need to use advanced techniques to set other properties (adding special code, or editing certain files).

Setting an application's properties is preferred to hard-coding that information in the code, say of an About box. Once set for the assembly, you can use those properties anywhere in your code (e.g., the text of an About box or splash screen) by retrieving them using the **`My.Application.Info`** object. For example:

```
ApplicationNameLbl.Text = My.Application.Info.ProductName
VersionLbl.Text = My.Application.Info.Version.ToString()
```

To set assembly properties, choose Properties from the Project menu, or open "My Project" from the Solution Explorer window.

There are several properties you can set from here. The most complex is the *version number*. All the assembly properties are stored in the text file `AssemblyInfo.vb`, which you can edit directly if you wish. You can also change the icon (browse for a `.ico` file) that shows for your application. (Note there are a number of free online tools that will convert .gif files to .ico files.)

Version Numbering in .net

The version number of an application ("Assembly version") typically consists of a series of numbers, each in the range 0 to 65534. One common scheme is to use *major.minor.revision*. The .net scheme uses 4 numbers *major.minor.build.revision* (see Version System), where:

- The **major number** only increments for a complete redesign of the code, a new user interface, or total new functionality. (Or, the marketing department insisted.) A new major number should be assigned if the new version is not compatible with the old version. When the major number changes the minor an revision numbers generally get reset to zero.
- The **minor number** increments for non-major changes that are user visible, such as huge bug has been fixed (or a set of them), a performance improvement, or some added or changed functionality that didn't warrant having the major number increment. Newer (higher) minor numbers (but the same major number) should indicate the new version is *backward compatible* with the previous version.
- The **build number** changes every time you make a public release. (That is, you re-compiled the code and selected the result to be make available.) You have different

build numbers when recompiling for a different platform, CPU type, or new compiler version. Typically these are based on the date of the release and can be a timestamp (number of seconds or milliseconds since 1/1/1970 00:00:00 GMT) or YYYYMMDD, or just some small number that you keep track of within your organization, so you know which date a build number indicates. (MS Office uses this scheme: MMDD, where MM is the number of months since the year the product was first released and DD is the day of the month. So for Office 2003 a build number of 3417 indicate a build on Oct. 17, 2005.)

- The **revision number** (a.k.a. patch number, bugfix number) increments every time changes are committed to the source code. "Committed" has a specific meaning here; this is an action done when using some source code control/revision/management system.

> Some applications use *major.minor.build-days.build-seconds* instead. Non-.net executables may use entirely different schemes.

When an application depends on another assembly, it will specify the assembly name and version number it needs (in a file `AssemblyInfo.vb`). If that version can't be found your application won't run. You can specify just a major (or major.minor) version number. See AssemblyVersionAttribute for details.

> To avoid having to re-build clients when you make minor (compatible) changes to some assembly, it is common to leave the minor, build, and revision numbers at zero all the time.

It is possible to have VS automatically set the build and revision number for you. You must specify the version number as "*major.minor.**" which causes the build number to be set to the number of days since 1/1/2000 (local time), and the revision number to be set to the number of seconds since midnight local time, divided by 2.

There are two version numbers you can set. The *Assembly Version* reflects the version of the specification of the assembly. This is most important when your assembly is a DLL. It changes when the API changes (types or methods added, modified or removed), or when the semantics of the API change (a method now does something functionally different). The *File Version* reflects the distribution. It changes when the binary image (the .exe or .dll) of the assembly changes, even when the Assembly Version does not. It is this version (plus the assembly name) that uniquely identifies the system to Windows.

While in theory the File Version allows any string to be used as a value, it is highly recommended to use the same four number version string scheme as for the Assembly Version number (and in fact, you can use the same value for both).

Version numbering schemes change over time and can be very confusing. Sun and other organizations have taken to have two version numbers for products, an internal number, and a number determined by the marketing department that is hopefully more user-friendly.

Creating a Splash Screen

A *splash screen* is a type of window that quickly loads and displays when an application is loading. This is useful since many applications can take a very long time to load and display a complete user interface. Displaying a splash screen during this time gives feedback to the user that the program is indeed starting.

Some splash screens include a progress bar to indicate the loading progress. This is especially useful when some application has a lot of plug-ins (add-ons) to load.

VB.net includes a template splash screen, a type of form only the window doesn't have a title bar or scroll bar. This standard splash screen includes the same sort of assembly information that commonly appears on an About window.

Once you add to your project and set up the splash screen the way you want, you need to set it to display first. You can select a form to use for a splash screen from the project properties page. This form displays while the "startup form" is being created and displayed. Once it is ready to go the splash screen automatically disappears.

For simple startup forms, the splash screen can disappear so quickly the user has no chance to see it. You can force the splash screen to stay visible for a minimum amount of time (say 3 or 4 seconds, or more) but doing so requires advanced techniques. Your book shows how to delay the closing of the splash screen, but if it has been visible for 10 seconds already you don't really want to keep it visible for another 5 seconds!

The correct technique is to start a timer when the splash screen loads. When the startup form is ready, the splash screen stays up until the timer expires; if it has already expired you close the splash screen at once. There is an Assembly property you can set for this, but it is tricky to do so. See My.Application.MinimumSplashScreenDisplayTime.

Working with Forms

Using forms works the same as for dialogs. You can show them either *modally* or *modelessly*, using **`Show`** (modeless) or **`ShowDialog`** (modal). (See discussion on page **Error! Bookmark not defined.**.)

You can hide a form with **`Hide`**. This sets the forms **`Visible`** property to `False`. (You can also change that to `True` or `False` to show and hide a form).

You can also use **`Close`**. If the form is shown modelessly `Close` completely removes it from memory. You would have to create a new instance of it with show to view it

again. But if a form is modal than `Close` is the same as `Hide`. so if you later make it visible again, the same form (with previous state —textbox contents, checkbox and radio button choices—intact).

Like all controls, **Forms have various events associated with them**. You can put code in the appropriate event handler to do something when a form is *Loaded* (created but not yet made visible), *Activated* (each time the form is made visible), *FormClosed* (occurs when the form is closed), and several others too.

> Forms have *lifetime* or *milestone* events that occur in order: Load, Activated, Paint, Deactivate, FormClosing, FormClosed.

To have the VS Designer tool create *stub* event handlers for forms is easy. Like all controls you just double-click on the from; this will create a Load event handler. To have VS create stubs for other event handlers, you view the code for some form. At the top-left, where is shows "General", select "`(NameOfForm Events)`". From the top-right, where it shows "(*`Declarations`*)", select the event handler you want to create.

You can also create these from the Designer window (for any control, not just forms). Select the control (the form), and in the properties window, click on the Events button (the one with the lightning bolt). You can then double-click on any event shown to create a stub handler for it.

Review of Scope and Lifetime ***(See page** Error! Bookmark not defined.**)***

Block/local scope: visible from the point of declaration to the end of the block it was declared in.

Class/module scope: visible throughout a class or module.

Name hiding: occurs when a local variable has the same name as a class/module one. Use `Me.`*`name`* to refer to a hidden class/module scoped variable.

Block/procedure lifetime: the variable/object is created when the flow of control passes through the declaration, and it is destroyed when the block/procedure end is reached.

Whole project lifetime: The variable/object is created when the project loads (roughly) and remains in memory until the project exits.

Using modifiers to change the scope, lifetime, and other attributes

Using `Static` to change lifetime: Normally block/local variables have block/procedure lifetime, and class/module variables have whole project lifetime. You can declare a local variable using Static to change its lifetime to whole project, but this doesn't change the scope.

Using access modifiers to change scope: While there is no way to change the scope of block/local variables, variables and procedures with class/module scope can be declared to make them visible from other classes/modules.

- Use **`Private`** to prevent such access from outside the current class/module,
- use **`Friend`** to allow access from any class/module in the same assembly, and
- use **`Public`** to allow access from anywhere. (Unless your project type is DLL, there's no real difference between `Friend` and `Public`.)

> There is also **`Protected`**, which allows a class scoped variable or procedure to be accessed from a derived class. (This requires inheritance, which will be discussed later.)

You can use `Protected` with `Friend`.

You should prefer to use `Private` unless you have a reason not to.

There are other modifiers too, that don't affect access, such as `Static` (remember this applies only to local variables), `ReadOnly`, `Const`, and `Shared` (applies only to class variables and will be discussed later.)

> Use the `ReadOnly` modifier to define a variable that needs to be initialized with a non-constant value, but should never change after initialization. The difference between `ReadOnly` properties and `Const` properties is when they are resolved. The value of a `Const` property is hard coded into the CIL at compile time. The value of a `ReadOnly` property is determined dynamically at runtime . Also, `ReadOnly` fields are per-instance (per-object) by default, whereas `Const` fields are implicitly `Shared`. If in doubt, prefer to use `Const`.

Namespaces

A namespace is hierarchical way to group modules and classes. All projects have a default, or ***root*** namespace (by default, the same name as the project itself). All classes and modules have a (fully) qualified name; so the class `Foo` in the namespace `Bar` has the qualified name `Bar.Foo`. Some of the standard .net classes and modules have very long qualified names.

To let the compiler know you will be referring to classes and modules in some namespace, you use the **`Imports`** *`namespace`* statement. This allow you to use the unqualified class/module name in your code. (Your project must include a *reference* to a namespace before it can be used. A number are added automatically from the VS template used. See the References tab on the My Project properties window.) The References tab on the project Properties window also allows you to define imported namespaces.

If you don't import the namespace, you will need to use the qualified names of classes and modules.

In VB.net 2008. you can also import classes and modules, to allow you to refer to their properties and procedures using unqualified names. You can also create *aliases*:

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

or:

```
Imports System
  Console.WriteLine("Hello")
```

or:

```
Imports System.Console
  WriteLine("Hello")
```

or:

```
Imports Con = system.Console
  Con.WriteLine("Hello")
```

Building Assemblies

When you build your project, an assembly is created. This is either a .exe or a .dll file. In either case the resulting executable depends on the .net runtime being installed. So if you send someone a copy of your wonderful .exe and they don't have the correct version of .net installed, it won't run.