Display a WinForms Splash Screen

by Juval Löwy

Technology Toolbox

- ☐ VB.NET
- ☑ C#
- ☐ SQL Server 2000
- ☐ ASP.NET
- ☐ XML
- ☐ VB6

Go Online!

Use these Locator+ codes at **www.visualstudiomagazine.com** to go directly to these resources.

Download
VS0304QA Download the code for this article. It includes the SplashScreen class.

Discuss
VS0304QA_D Discuss this article in the C# forum.

Read More
VS0304QA_T Read this article online.

VSEP011129JL_T "System Programming: Object Serialization and Persistence" by Juval Löwy

VS0302QA_T Q&A, "Call WinForms on Multiple Threads," by Juval Löwy and Karl E. Peterson

VS0201GS_T Getting Started, "Understand Visual Inheritance," by Stan Schultes

Q: Display a WinForms Splash Screen

My application takes a while to start up. I want to display a splash screen (as Visual Studio .NET and Office applications do) while the application continues loading in the background. The toolbox has no such control. How do I implement it?

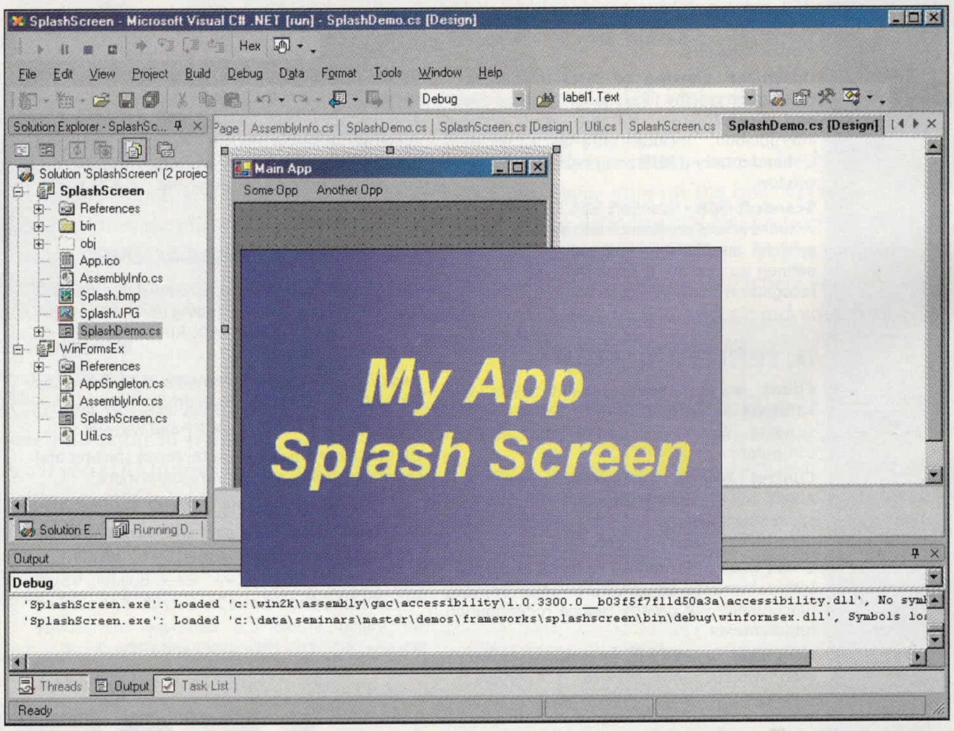
A:

The code accompanying this column contains the SplashScreen class (download the code from the *VSM* Web site; see the Go Online box for details):

```
public class SplashScreen
{
   public SplashScreen(Bitmap splash);
   public void Close();
}
```

SplashScreen's constructor accepts the bitmap to display. The Close method closes the splash screen. You typically use SplashScreen in the method that handles your form's Load event (see the resulting splash screen in Figure 1):

```
private void OnLoad(object
   sender,EventArgs e)
{
   Bitmap splashImage;
   splashImage = new
      Bitmap("Splash.bmp");

   SplashScreen splashScreen;
   splashScreen = new
```



```
'SplashScreen.exe': Loaded 'c:\win2k\assembly\gac\accessibility\1.0.3300.0__b03f5f7f11d50a3a\accessibility.dll', No sym
'SplashScreen.exe': Loaded 'c:\data\seminars\master\demos\frameworks\splashscreen\bin\debug\winformsex.dll', Symbols lo
```

Figure 1 Create a Splash Screen. When you start the demo application, you see the splash screen, followed by the application's main form. The splash screen is a top-most window, rendered by a dedicated thread. As a result, it repaints itself even if you swap another application into context.

```
SplashScreen(splashImage);
  //Do some lengthy operations, then:
  splashScreen.Close();
  Activate();
}
```

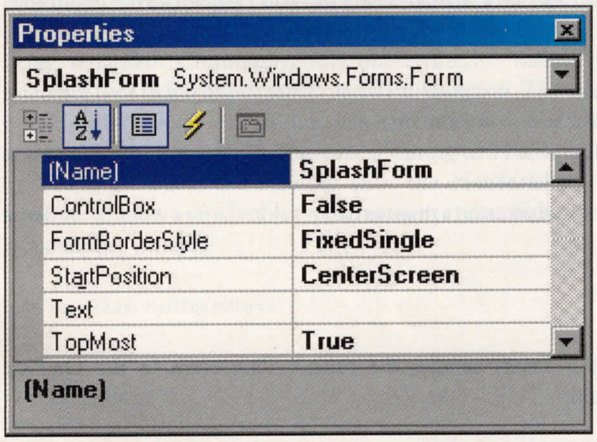
After you close the splash screen, you activate your form in order to bring it to the foreground and give it focus.

You can use any bitmap as a splash screen. You can create the bitmap from a BMP or JPG file by constructing a new bitmap object around it:

```
Bitmap splashImage;
splashImage = new Bitmap("Splash.bmp");
```

Or, you can use an image you load from the form's resources:

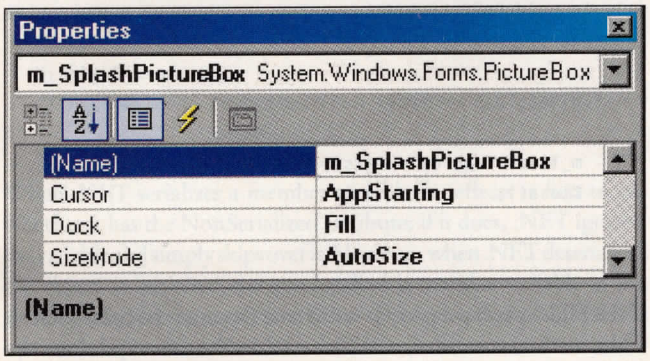
```
using System.Resources;

ResourceManager resources;
resources = new
  ResourceManager(typeof(MyForm));

Bitmap splashImage;
```



Properties
SplashForm System.Windows.Forms.Form

(Name)	SplashForm
ControlBox	False
FormBorderStyle	FixedSingle
StartPosition	CenterScreen
Text	
TopMost	True

(Name)



Properties
m_SplashPictureBox System.Windows.Forms.PictureBox

(Name)	m_SplashPictureBox
Cursor	AppStarting
Dock	Fill
SizeMode	AutoSize

(Name)

Figure 2 Set Visual Properties for the Splash Form and the Picture Box. You can use the visual designer to achieve almost everything you need from a visual standpoint to transform a standard form to a splash screen. Use SplashForm's and m_SplashPictureBox's Properties windows to set new values that determine control box, border, start position, caption, position, cursor, sizing, and docking.

```
SplashImage =
  (Bitmap)(resources.GetObject(
  "SplashImage"))
```

There's more to implementing a splash screen than meets the eye. It relies on some nifty WinForms features, and it involves some interesting design issues that apply in other WinForms situations. The splash screen is actually a WinForms form called SplashForm. You can use the WinForms visual designer to make most of the changes required to transform a default form into a splash screen—a testament to how versatile, yet easy, WinForms are. In this implementation, you add a single control to the form—a simple picture box called m_SplashPictureBox.

You don't know the splash image's size at compile time, because it's a runtime parameter, yet the picture box needs to size itself up to it. You can do this easily by setting the m_SplashPictureBox's SizeMode property to AutoSize. Next, you must align the picture box at the top-left corner of the form. You can do this by setting m_SplashPictureBox's Dock property to Fill. This snaps it to the top-left corner. It'll expand toward the right-bottom corner at run time to fill its containing form, because the size mode is set to AutoSize. Finally, set m_SplashPictureBox's Cursor property to AppStarting (an hourglass plus a pointer), so that if the user moves the mouse over the splash screen, he or she is aware that the application is starting up.

The splash form itself shouldn't display any control-box buttons (the close, minimize, and maximize boxes), nor should it have a caption bar. Use the visual designer to set SplashForm's ControlBox property to False; this removes the control box. Clear the Text property in the designer to remove the caption bar.

The splash screen's border is next. It should be a single line—not

C# • Implement the SplashScreen Class

```
public class SplashScreen
{
  public SplashScreen(Bitmap splash)
  {
    m_SplashImage = splash;
    ThreadStart threadStart = new
      ThreadStart(Show);
    m_WorkerThread = new Thread(threadStart);
    m_WorkerThread.Start();
  }
  void Show()
  {
    m_SplashForm = new SplashForm(m_SplashImage);
    m_SplashForm.ShowDialog();
  }
  public void Close()
  {
    m_SplashForm.HideSplash = true;
    m_WorkerThread.Join();
  }
  Bitmap m_SplashImage;
  SplashForm m_SplashForm;
  Thread m_WorkerThread;
}
```

Listing 1 SplashScreen's constructor creates a thread to call the Show method, which displays the SplashForm form. Show passes SplashForm the image to splash, then calls ShowDialog to display the form. The Close method signals SplashForm to hide and waits for the worker thread to terminate.

the default sizable border style—so set the form's FormBorderStyle property to FixedSingle. Set the TopMost property to True to have to splash screen always at the top of the z-order (the order in which Windows displays windows on the desktop). A splash screen should always be at the center of the screen. Fortunately, you can set the StartPosition property to CenterScreen, and WinForms takes the window's size into account automatically and centers it. Figure 2 shows the Properties windows of both SplashForm and m_SplashPictureBox, summarizing the properties you need to set and the new values.

Next, write some code to size up the splash screen. SplashForm's constructor accepts the image to splash and assigns it to the picture box's image:

```
internal class SplashForm : Form
{
  PictureBox m_SplashPictureBox;
  public SplashForm(Bitmap
    splashImage)
  {
    InitializeComponent();
    m_SplashPictureBox.Image =
      splashImage;
    ClientSize =
        m_SplashPictureBox.Size;
  }
  //Rest of the implementation
}
```

Note that you must set SplashForm's client size to that of the picture box, which sizes itself up to the image's size automatically. The result is that the SplashForm now displays the image in the picture box exactly, because the picture box is aligned at the top-left corner of the form.

You can't display SplashForm on the same thread you use to load the application, because that thread is busy loading the app and will never get around to displaying or repainting the splash screen. Instead, have SplashScreen create a worker thread to display SplashForm (see Listing 1). The worker thread calls the Show method, which creates the SplashForm object and calls its ShowDialog method:

```
void Show()
{
  m_SplashForm = new
    SplashForm(m_SplashImage);
  m_SplashForm.ShowDialog();
}
```

ShowDialog displays the form and starts pumping Windows messages to it. The splash screen runs on its own thread, so that thread is doing the message processing—not the main application thread that's busy loading the application.

The next challenge is finding a way for the main application to close the splash screen. The easiest way would be to signal the worker thread to close the form—except the thread method (Show) is busy pumping messages in the form's message loop (the ShowDialog method) and isn't available to check a flag or an event. The solution is simple—use Windows Timers. Use the designer to add a Timer control to the form, and set its Interval property to some adequate value, such as 500 milliseconds. The Timer class is actually based on the WM_TIMER message, so the timer's Tick event is Windows-message–driven. The worker thread pumps that message to the splash screen, where it checks whether it needs to close the splash screen, because the main application has finished loading. The SplashForm class provides the Boolean property HideSplash, which SplashScreen's Close method sets:

```
public void Close()
{
  m_SplashForm.HideSplash = true;
  m_WorkerThread.Join();
}
```

HideSplash provides access to the m_HideSplash Boolean member variable of SplashForm. m_HideSplash is being accessed by multiple threads, so HideSplash needs to provide access to m_HideSplash in a thread-safe access by locking the SplashForm:

```
public bool HideSplash
{
  get
  {
    lock(this){
      return m_HideSplash;
    }
  }
  set
  {
    lock(this){
      m_HideSplash = value;
    }
  }
}
```

SplashForm handles the timer's Tick event in the OnTick method:

```
private void OnTick(object
  sender,EventArgs e)
{
  if(HideSplash == true)
  {
    m_Timer.Enabled = false;
    Close();
  }
}
```

If the HideSplash property is set to true (because the SplashScreen Close method was called), OnTick disables the timer and closes the SplashForm. It all works together like this: The main form starts loading, and it displays the splash screen on a different thread. The main form then continues with the application startup. The splash screen checks periodically (using the timer) whether it should close. The main form calls SplashForm's Close method when it's done loading. The Close method sets HideSplash to true and calls Join on the worker thread, waiting for it to terminate. This blocks the main form, so it doesn't display itself as long as the splash screen is displayed. The next time the timer ticks, it checks the value of HideSplash. It cancels the timer and closes SplashForm, because HideSplash is set to true. This causes the ShowDialog method (called in the Show method of SplashScreen) to return, and then Show returns. The thread is terminated once Show returns, because Show is the worker thread's thread method. At this point, the call to Join in the Close method of SplashScreen returns. The Close method returns control to the main form, which now displays itself.

Q: Allow Serializable Types to Contain Nonserializable Members

I have a serializable class that contains a database connection as a member variable. I get an exception when I try to serialize the class, because the connection isn't serializable. If I mark the connection as nonserializable, then I can serialize—but I can't use the object after deserialization, because the connection member is invalid. What can I do?

A:

When you use the Serializable attribute to mark a class for serialization, .NET insists that all its member variables be serializable as well, and throws an exception of type SerializationException during serialization if it discovers a nonserializable member. However, the class might have a member that can't be serialized. This type doesn't have the Serializable attribute and prevents the containing type from being serialized. Commonly, this nonserializable member is a reference type that requires some special initialization. The solution to this problem requires marking such a member as nonserializable and taking a custom step to initialize it during deserialization.

You must mark the member with the NonSerialized field attribute to allow a serializable type to contain a nonserializable type as a member variable:

```
public class MyOtherClass
{..}

[Serializable]
public class MyClass
{
   [NonSerialized]
   MyOtherClass m_Obj;
   /* Methods and properties */
}
```

When .NET serializes a member variable, it reflects it first to see whether it has the NonSerialized attribute; if it does, .NET ignores the variable and simply skips over it. However, when .NET deserializes the object, it initializes the nonserializable member variable to the default value for that type (a null for all reference types). Then it's up to you to provide code to initialize the variable to its correct value. To that end, the object must know when it's being deserialized. You must implement the IDeserializationCallback interface, defined in the System.Runtime.Serialization namespace:

```
public interface
   IDeserializationCallback
{
   void OnDeserialization(object
      sender);
}
```

.NET calls IDeserializationCallback's single OnDeserialization() method after .NET has finished deserializing the object, allowing it to perform the required custom initialization steps. You can ignore the sender parameter because .NET always sets it to null. This code demonstrates how you can perform custom serialization by implementing IDeserializationCallback:

```
using System.Runtime.Serialization;

[Serializable]
public class MyClass :
   IDeserializationCallback
{
   [NonSerialized]
   IDbConnection m_Connection;

   public void OnDeserialization(object
      sender)
   {
      Debug.Assert(m_Connection ==
         null);
      m_Connection = new
         SqlConnection();
      m_Connection.ConnectionString =
         "data
         source= ... ";
      m_Connection.Open();
   }
   /* Other members */
}
```

The MyClass class in the preceding code has a database connection as a member variable. The connection object (SqlConnection) isn't a serializable type, so you mark it with the NonSerialized attribute. MyClass creates a new connection object in its implementation of OnDeserialization(), because the connection member is set to its default value of null after deserialization. MyClass then initializes the connection object by providing it with a connection string, and opens it. VSM

Juval Löwy is a software architect and the principal of IDesign, a consulting and training company focused on .NET design and .NET migration. Juval is Microsoft's regional director for the Silicon Valley, working with Microsoft on helping the industry adopt .NET. His latest book is *Programming .NET Components* (O'Reilly & Associates). Juval speaks frequently at software-development conferences. Contact him at www.idesign.net.

Additional Resources

Programming .NET Components by Juval Löwy [O'Reilly & Associates, 2003, ISBN: 0596003471]