CHAPTER 2

Creating the Sudoku Application

Now that you have a firm grounding in the basic rules of Sudoku, it is time for us to start the journey into solving Sudoku puzzles using computer programming. For this task, you will build a Windows application that represents a Sudoku puzzle. The application that you build in this chapter will act as a rule enforcer, helping you to make sure that a value inserted into a cell does not violate the rules of Sudoku. We aren't concerned about how to solve a Sudoku puzzle yet; we leave that for the next few chapters.

In this chapter, I walk you through the various steps to construct a Sudoku puzzle board using a Windows application. This is the foundation chapter that all future chapters will build on. While the application that you build in this chapter lacks the intelligence required to solve a Sudoku puzzle, it will provide you with many hours of entertainment. Moreover, it will provide some aid to beginning Sudoku players, because it helps to check for the rules of Sudoku. Your Sudoku application will have the capabilities to do the following:

- Load and save Sudoku puzzles
- Ensure that only valid numbers are allowed to be placed in a cell
- Check whether a Sudoku puzzle has been solved
- Keep track of the time needed to solve a Sudoku puzzle
- Undo and redo previous moves

As in all large software projects, I will be breaking the functionalities of the Sudoku application into various functions and subroutines. The following are the major tasks in this chapter:

- Creating the user interface of the Sudoku application
- Using arrays to represent values in the grid
- Storing the moves using the stack data structure
- Generating the grid dynamically using Label controls
- Handling click events on the Label controls
- Checking whether a move is valid
- Checking whether a puzzle is solved
- Updating the value of a cell
- Undoing and redoing a move
- Saving a game
- Opening a saved game
- Ending the game

At the end of this chapter, you will have a functional Sudoku application that you can use to solve your Sudoku puzzles!

Creating the Sudoku Project

The application that you will build in this chapter is a Windows application. Figure 2-1 shows how the application will look at the end of this chapter.

Using this application, users will be able to load and save puzzles to disk. The application will act as a rule enforcer, ensuring that the user cannot place a number in a cell that will violate the rules of Sudoku. This is useful for beginners who are learning Sudoku.

■ **Note** The application in this chapter will not have the intelligence to solve a Sudoku puzzle yet. You will begin building the intelligence in Chapter 3.

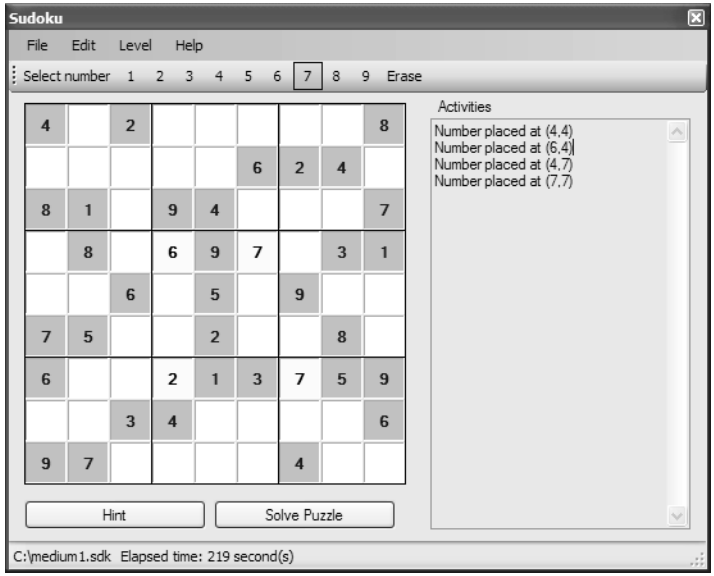


Figure 2-1. *The Sudoku application you will build in this chapter*

Creating the User Interface

For the Sudoku application, you will create a Windows application using Microsoft Visual Studio 2005. Launch Visual Studio 2005. Choose File ➤ New Project, select the Windows Application template, and name the project **Sudoku**.

■ **Note** Throughout this book, I will use Visual Basic 2005 as the programming language. C# programmers should not have any major problem understanding/translating the code.

The project contains a default Windows form named `Form1`. Set the properties of `Form1` as shown in Table 2-1. To change the property of a control in Visual Studio 2005, right-click the control and select Properties to open the Properties window.

Table 2-1. *Properties of Form1*

Property	Value
FormBorderStyle	FixedToolWindow
Size	551, 445
Text	Sudoku

Figure 2-2 shows how `Form1` will look like after applying the properties listed in Table 2-1. Essentially, you are creating a fixed-size window.

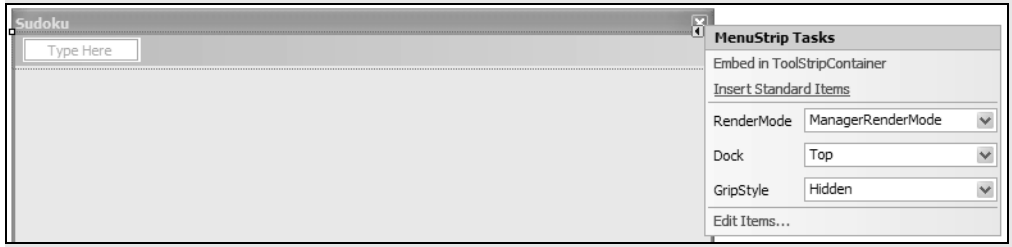
Figure 2-2. *Modifying Form1*

Adding a MenuStrip Control

In the Toolbox, double-click the MenuStrip control located on the Menus & Toolbars tab to add a menu to `Form1`. In the MenuStrip Tasks menu (also known as a Smart Tag), click Insert Standard Items to insert a list of standard menu items.

SMART TAGS IN VISUAL STUDIO 2005

A Smart Tag is a panel that is displayed next to a control (by clicking the arrow icon at the top-right corner of the control), containing a list of commonly used properties. By saving you a trip to the Properties window for some of the more common properties you need to set, Smart Tags can improve development productivity. Smart Tags are a new feature in Visual Studio 2005.



Once the standard menu items are inserted, you can customize the menu by removing menu items that are not relevant (use the Delete key to remove menu items) and inserting new items. Figure 2-3 shows the different menu items that you will add for this application.

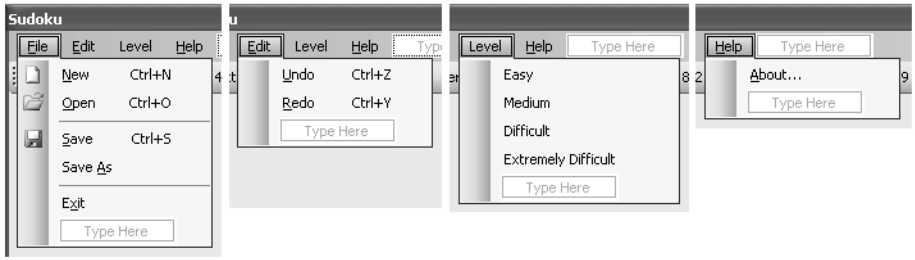


Figure 2-3. *The menu items for the Sudoku application*

■ **Tip** The standard menus by default include a Tools menu rather than a Level menu. You can simply replace the Tools menu with the Level menu. In addition, you can change the menu items to Easy, Medium, Difficult, and Extremely Difficult. For the File, Edit, and Help menus, if you want to delete any menu items, simply select the unwanted item and press the Delete key.

To assign shortcuts to the different levels of difficulty, click each of the Level menu items and enter the values as shown in Table 2-2 (see also Figure 2-4).

Table 2-2. *Values to Set for the Level Menu and Its Menu Items*

Menu/Item	Value
Level	&Level
Easy	&Easy
Medium	&Medium
Difficult	&Difficult
Extremely Difficult	Ex&tremely Difficult

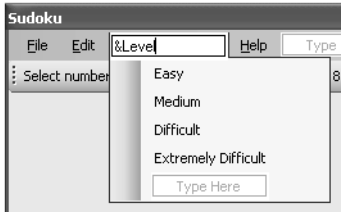


Figure 2-4. *Setting the values for the Level menu and its menu items*

After you set the values, the Level menu looks like Figure 2-5.

Figure 2-5. *The Level menu and its menu items*

Adding a ToolStrip Control

You will now add a ToolStrip control to the Windows form so that users can choose a number to insert into the cells. In the Toolbox, double-click the ToolStrip control (also located on the Menus & Toolbars tab) to add it onto `Form1`. You need to add Label and Button controls to the ToolStrip control; Figure 2-6 shows how to add controls to a ToolStrip control.

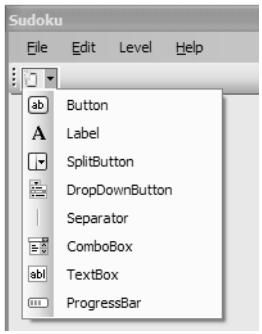


Figure 2-6. *Adding controls to the ToolStrip control*

Add a Label control to the ToolStrip control, and set the `Text` property of the Label control to **Select number**.

Next, add ten Button controls to the ToolStrip control. Set the `DisplayStyle` property of each Button control to Text. Set the `Text` property of the ten Button controls to **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, and **Erase**, respectively, as shown in Figure 2-7, which depicts how the finished ToolStrip control should look.

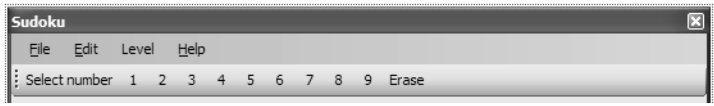


Figure 2-7. *The finished ToolStrip control*

Adding a StatusStrip Control

You will also add to the bottom of the form a StatusStrip control (also located on the Menus & Toolbars tab). Click the StatusStrip control on the form and insert two StatusLabel controls (see Figure 2-8).

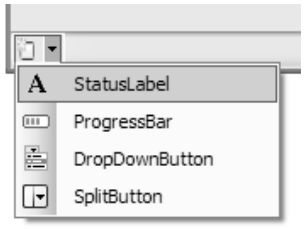


Figure 2-8. *Populating the StatusStrip control*

Figure 2-9 shows the form at this stage.

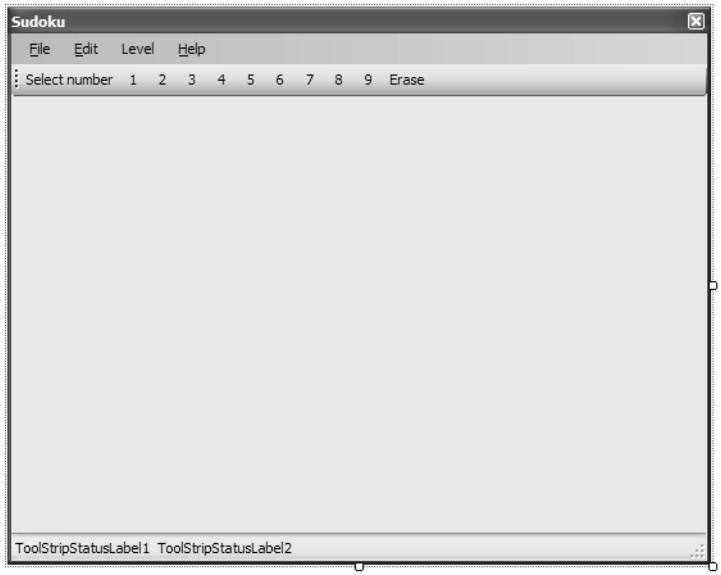


Figure 2-9. *The form with the various menu controls*

Adding Other Controls

The last step in creating the graphical user interface (GUI) of the Sudoku application is to add the various controls, as shown in Figure 2-10.

■ **Note** What about drawing the Sudoku grid? Well, I will be using Label controls to represent the cells within a Sudoku grid. And since there are 81 of them, I will generate them dynamically. I will show you how to do this in the next section.

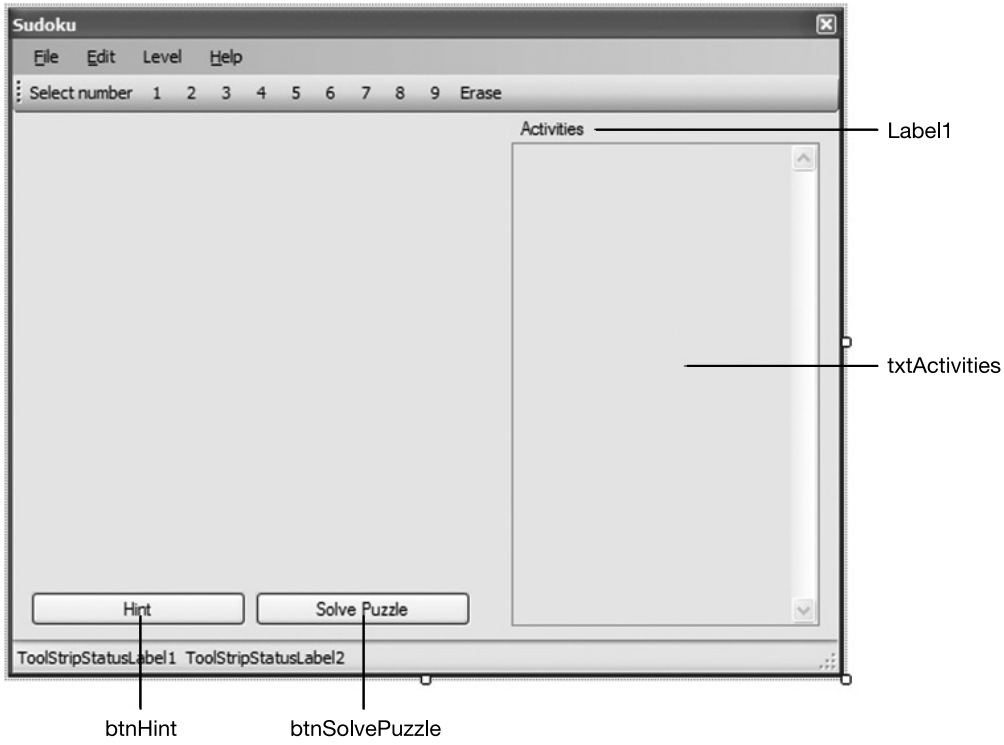


Figure 2-10. *Adding the various controls to Form1*

The txtActivities control is used to display the various moves played by the user. Set the properties of these controls as shown in Table 2-3.

Finally, add a Timer control (located on the Components tab in the Toolbox) to the form. Set its `Interval` property to `1000` (the unit is in milliseconds). The Timer control is used to keep track of the time taken to solve a Sudoku puzzle.

Table 2-3. *Properties of the Various Controls*

Control	Property	Value
Label (Label1)	Location	332, 53
Label (Label1)	Text	Activities
TextBox (txtActivities)	Location	329, 69
TextBox (txtActivities)	Multiline	True
TextBox (txtActivities)	Size	203, 321
TextBox (txtActivities)	Scrollbars	Vertical
Button (btnHint)	Text	Hint
Button (btnHint)	Location	12, 367
Button (btnHint)	Size	142, 23
Button (btnSolvePuzzle)	Text	Solve Puzzle
Button (btnSolvePuzzle)	Location	160, 367
Button (btnSolvePuzzle)	Size	142, 23

Declaring the Member Variables

Now that you have added the various controls to the form, it is time to switch to the code-behind of `Form1` to add the various functionalities. In Solution Explorer, select `Form1.vb` and click the View Code button to switch to the code-behind of `Form1` (see Figure 2-11).

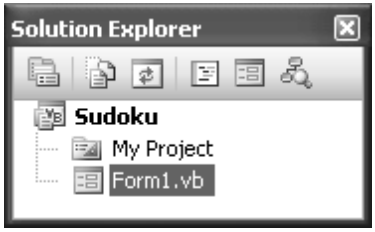


Figure 2-11. *Switching to code view*

In the `Form1` class, add the following member variables (in bold):

```
Public Class Form1

    '---dimension of each cell in the grid---
    Const CellWidth As Integer = 32
    Const cellHeight As Integer = 32
```

```
'---offset from the top-left corner of the window---
Const xOffset As Integer = -20
Const yOffset As Integer = 25

'---color for empty cell---
Private DEFAULT_BACKCOLOR As Color = Color.White

'---color for original puzzle values---
Private FIXED_FORECOLOR As Color = Color.Blue
Private FIXED_BACKCOLOR As Color = Color.LightSteelBlue

'---color for user-inserted values---
Private USER_FORECOLOR As Color = Color.Black
Private USER_BACKCOLOR As Color = Color.LightYellow

'---the number currently selected for insertion---
Private SelectedNumber As Integer

'---stacks to keep track of all the moves---
Private Moves As Stack(Of String)
Private RedoMoves As Stack(Of String)

'---keep track of filename to save to---
Private saveFileName As String = String.Empty

'---used to represent the values in the grid---
Private actual(9, 9) As Integer

'---used to keep track of elapsed time---
Private seconds As Integer = 0

'---has the game started?---
Private GameStarted As Boolean = False
```

As you can see from the declaration, you first declared some constants to store the dimension of each cell in the Sudoku grid. You also declared some variables to store the various colors in the grid—all original values in the grid will have a blue background, while values placed by the user will have a yellow background. Empty cells have a white background.

Next, you declared two stack data structures—`Moves` and `RedoMoves`. A stack is a data structure that works on the last-in, first-out (LIFO) principle. This means that the last item pushed into a stack is the first item to be taken off. You use the `Stack` class to remember

the moves you made so that if you need to undo the moves, you can do so. I will discuss this issue in more detail later in the chapter, in the section "Storing Moves in Stacks."

If you observe the declaration of the stack, you will notice that there is a new keyword, `Of`. This keyword is used when declaring a *generic* type. Support for generic types is a new feature in .NET Framework 2.0. In our case, the `Stack` class is a generic class. During declaration time, you use the `Of` keyword to indicate to the compiler that you can only push and pop string data types (and not other data types) into and from the `Stack` class. This helps to make your application safer and reduces the chance that you inadvertently push or pop the wrong types of data into the stack.

Representing Values in the Grid

A standard Sudoku puzzle consists of a grid of nine rows and nine columns, totaling 81 cells. A good way to represent a Sudoku grid is to use a two-dimensional array. As an example, the grid in Figure 2-12 will be represented in the array as follows (recall that a cell in a Sudoku puzzle is referenced by its column number followed by its row number):

```
actual(1,1) = 4
actual(2,1) = 0
actual(3,1) = 2
actual(4,1) = 0
actual(5,1) = 3
...
actual(1,2) = 7
...
```

Each empty cell in the grid is represented by the value 0.

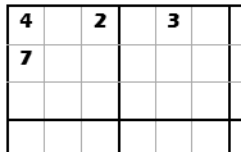


Figure 2-12. *Representing cells in a Sudoku grid using an array*

However, note that arrays in Visual Basic 2005 are zero-based. That is, when you declare the actual variable to be `actual(9,9)`, there are actually 100 elements in it, from `actual(0,0)` to `actual(9,9)`. For our application, the elements in row 0 and column 0 are left unused, as shown in Figure 2-13.

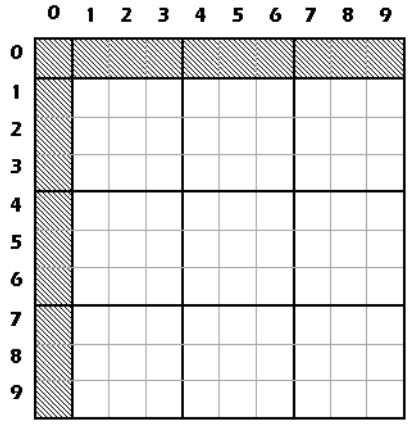


Figure 2-13. *Unused cells (shaded) in the array*

Naming Cells

Each cell in the grid will be represented using a dynamically generated Label control. You will need to assign a name to each Label control so that individual cells can be identified. For simplicity, in our application each cell will be identified based on its column and row numbers. For example, the Label control representing cell (1,1) will be named 11, cell (2,1) will be named 21, and so on.

Erasability of a Cell

A cell may contain a value set by the user or set originally as part of the puzzle. If the value is set by the user, it can be erased so that other values can be assigned to it. As such, there must be a way to identify if a particular cell value can be erased. For this purpose, you can use the `Tag` property of the Label control. As an example, if the value in cell (4,5) can be erased, you will set its `Tag` property to `1`. If its value cannot be erased, then its `Tag` property would be `0`.

Storing Moves in Stacks

To allow the user to undo and redo his moves, every time a number is placed in a cell, its coordinates and values are placed in a stack. When the user undoes his move, a value is popped from the stack and pushed into another stack. The value pushed into the stack is a three-digit string. For example, 349 means that cell (3,4) has been assigned the value 9. Figure 2-14 shows that when a user undoes a move, the value from the `Moves` stack is popped and pushed into the `RedoMoves` stack. Similarly, when the user redoes a move, a value is popped from the `RedoMoves` stack and re-pushed into the `Moves` stack.

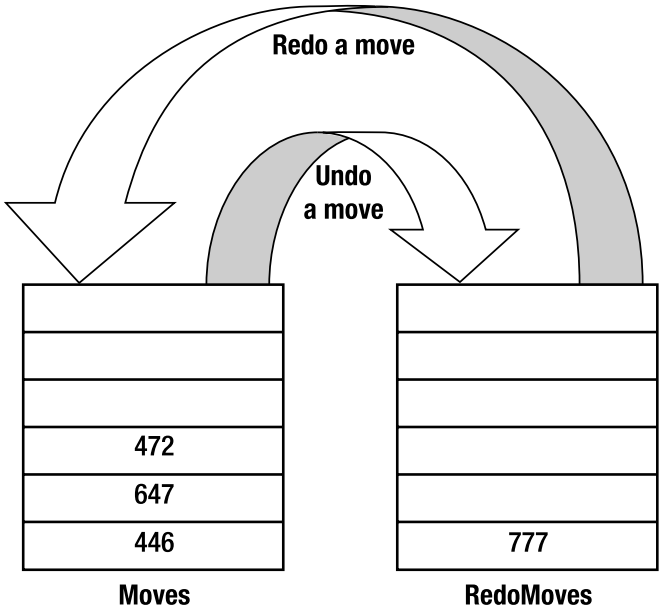


Figure 2-14. *Using stacks for undo and redo options*

Generating the Grid Dynamically

The first thing to do when the application loads is to generate the grid of a Sudoku puzzle. The `DrawBoard()` subroutine dynamically creates 81 Label controls to represent each cell in the 9×9 grid:

```
'================================================
' Draw the cells and initialize the grid
'================================================
Public Sub DrawBoard()
    '---default selected number is 1---
    ToolStripButton1.Checked = True
    SelectedNumber = 1

    '---used to store the location of the cell---
    Dim location As New Point
    '---draws the cells
    For row As Integer = 1 To 9
        For col As Integer = 1 To 9
            location.X = col * (CellWidth + 1) + xOffset
            location.Y = row * (cellHeight + 1) + yOffset
            Dim lbl As New Label
```

```
                With lbl
                    .Name = col.ToString() & row.ToString()
                    .BorderStyle = BorderStyle.Fixed3D
                    .Location = location
                    .Width = CellWidth
                    .Height = cellHeight
                    .TextAlign = ContentAlignment.MiddleCenter
                    .BackColor = DEFAULT_BACKCOLOR
                    .Font = New Font(.Font, .Font.Style Or _
                            FontStyle.Bold)
                    .Tag = "1"
                    AddHandler lbl.Click, AddressOf Cell_Click
                End With
                Me.Controls.Add(lbl)
            Next
        Next
    End Sub
```

Note that as you type the line `AddHandler lbl.Click, AddressOf Cell_Click`, you will get a compiler error, because the method has not been defined yet. For now, let's add an empty `Cell_Click()` method stub so that the compiler does not complain:

```
Private Sub Cell_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs)
    '---content to be populated later---
End Sub
```

Each Label control is hooked to the `Cell_Click()` event handler, which is fired when the user clicks each Label control (we will declare in it a later section).

The board is first drawn when the form loads, in the `Form1_Load()` event (you can simply double-click an empty portion of `Form1` to create this event handler):

```
Private Sub Form1_Load( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load
    '---initialize the status bar---
    ToolStripStatusLabel1.Text = String.Empty
    ToolStripStatusLabel2.Text = String.Empty
    '---draw the board---
    DrawBoard()
End Sub
```

Figure 2-15 shows what the form looks like when it loads.

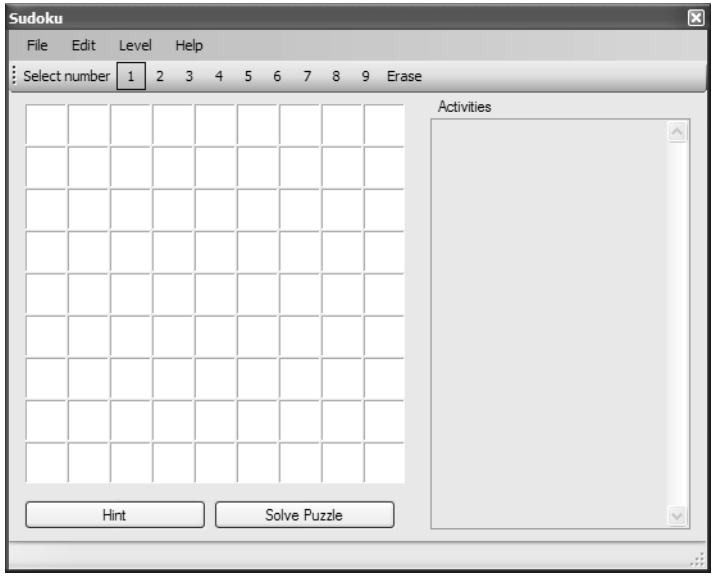


Figure 2-15. *Dynamically generating the Label controls*

One feature is missing, however. In a Sudoku puzzle, nine minigrids are contained within the bigger grid. You need a way to outline the nine minigrids. You do that by actually drawing the lines—four horizontally and four vertically. The `Form1_Paint()` event is the event that you use to insert the code to draw the eight lines:

```
'================================================
' Draw the lines outlining the minigrids
'================================================
Private Sub Form1_Paint( _
   ByVal sender As Object, _
   ByVal e As System.Windows.Forms.PaintEventArgs) _
   Handles Me.Paint

    Dim x1, y1, x2, y2 As Integer
    '---draw the horizontal lines---
    x1 = 1 * (CellWidth + 1) + xOffset - 1
    x2 = 9 * (CellWidth + 1) + xOffset + CellWidth
    For r As Integer = 1 To 10 Step 3
        y1 = r * (cellHeight + 1) + yOffset - 1
        y2 = y1
        e.Graphics.DrawLine(Pens.Black, x1, y1, x2, y2)
    Next
```

```
        '---draw the vertical lines---
        y1 = 1 * (cellHeight + 1) + yOffset - 1
        y2 = 9 * (cellHeight + 1) + yOffset + cellHeight
        For c As Integer = 1 To 10 Step 3
            x1 = c * (CellWidth + 1) + xOffset - 1
            x2 = x1
            e.Graphics.DrawLine(Pens.Black, x1, y1, x2, y2)
        Next
    End Sub
```

Figure 2-16 shows the effect of drawing these eight lines on the grid.

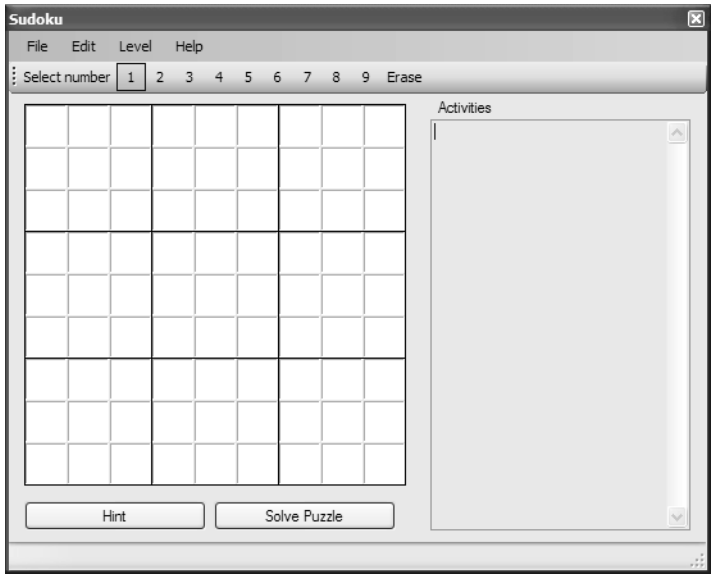


Figure 2-16. *The grid with the eight lines*

Starting a New Game

To start a new game, the user will select File ➤ New. For now, you will simply clear the board and reset a few variables. In Chapter 6, you will be more adventurous and learn how to generate a new Sudoku puzzle of varying levels of difficulty.

When a user starts a new game, be sure to ask if she wants to save the current game. If she does, save the current game before you start a new game. To add an event handler for the New menu item, double-click the New menu item in design view of Visual Studio and the event handler for the New menu item will appear. Code the following:

```
'================================================
' Start a new game
'================================================
Private Sub NewToolStripMenuItem_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles NewToolStripMenuItem.Click

    If GameStarted Then
        Dim response As MsgBoxResult = _
           MessageBox.Show("Do you want to save current game?", _
                           "Save current game", _
                           MessageBoxButtons.YesNoCancel, _
                           MessageBoxIcon.Question)

        If response = MsgBoxResult.Yes Then
            SaveGameToDisk(False)
        ElseIf response = MsgBoxResult.Cancel Then
            Return
        End If
    End If

    StartNewGame()
End Sub
```

As usual, to prevent the compiler from complaining about the missing `SaveGameToDisk()` subroutine, add a stub for this subroutine:

```
Public Sub SaveGameToDisk(ByVal saveAs As Boolean)
    '---content to be populated later---
End Sub
```

The `StartNewGame()` subroutine simply resets a few variables and updates a Label control located in the status bar. It also calls the `ClearBoard()` subroutine, which clears the values in the grid. The code follows:

```
'================================================
' Start a new game
'================================================
 Public Sub StartNewGame()
    saveFileName = String.Empty
    txtActivities.Text = String.Empty
    seconds = 0
    ClearBoard()
```

```
        GameStarted = True
        Timer1.Enabled = True
        ToolStripStatusLabel1.Text = "New game started"
    End Sub
```

The `ClearBoard()` subroutine prepares the Sudoku grid for a new game and creates a new instance of the `Moves` and `RedoMoves` stack objects:

```
    '================================================
    ' Draws the board for the puzzle
    '================================================
    Public Sub ClearBoard()
        '---initialize the stacks---
        Moves = New Stack(Of String)
        RedoMoves = New Stack(Of String)

        '---initialize the cells in the board---
        For row As Integer = 1 To 9
            For col As Integer = 1 To 9
                SetCell(col, row, 0, 1)
            Next
        Next
    End Sub
```

Notice that when a new game is started, the Timer control is also enabled so that the clock can start running to keep track of the time elapsed. The `Timer1_Click()` event is fired every 1 second (which is equivalent to 1000 milliseconds, as set in the `Interval` property). The elapsed time is displayed in the Label control located in the status bar. To display the elapsed time, add the following event to your code:

■ **Tip** Double-click the Timer control at the bottom of `Form1` to reveal this code-behind.

```
    '================================================
    ' Increment the time counter
    '================================================
    Private Sub Timer1_Tick( _
       ByVal sender As System.Object, _
       ByVal e As System.EventArgs) Handles Timer1.Tick
        ToolStripStatusLabel2.Text = "Elapsed time: " & _
                                     seconds & " second(s)"
        seconds += 1
    End Sub
```

Selecting the Numbers to Insert

Once a new game is started, the user will select a number to insert into the cells. You need to ensure that only one number is selected in the toolbar. The `SelectedNumber` variable keeps track of which number is currently selected, and if the user clicks the Erase button, the number is saved as a 0. To highlight the number selected by the user in the toolbar, create the `ToolStripButton_Click()` event:

```
'================================================
' Event handler for the ToolStripButton controls
'================================================
Private Sub ToolStripButton_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles _
   ToolStripButton1.Click, _
   ToolStripButton2.Click, _
   ToolStripButton3.Click, _
   ToolStripButton4.Click, _
   ToolStripButton5.Click, _
   ToolStripButton6.Click, _
   ToolStripButton7.Click, _
   ToolStripButton8.Click, _
   ToolStripButton9.Click, _
   ToolStripButton10.Click

    Dim selectedButton As ToolStripButton = _
       CType(sender, ToolStripButton)

    '---uncheck all the Button controls in the ToolStrip---
    '---ToolStrip1.Items.Item(0) is "Select Number"
    '---ToolStrip1.Items.Item(1) is "1"
    '---ToolStrip1.Items.Item(2) is "2", etc
    '---ToolStrip1.Items.Item(10) is "Erase", etc
    For i As Integer = 1 To 10
        CType(ToolStrip1.Items.Item(i), ToolStripButton).Checked = False
    Next

    '---set the selected button to "checked"---
    selectedButton.Checked = True
```

```
        '---set the appropriate number selected---
        If selectedButton.Text = "Erase" Then
            SelectedNumber = 0
        Else
            SelectedNumber = CInt(selectedButton.Text)
        End If
End Sub
```

Notice that the `ToolStripButton_Click()` event handles multiple events. You can make it handle multiple events by separating with commas the events of each control that you want to handle.

Figure 2-17 shows a number selected in the toolbar.

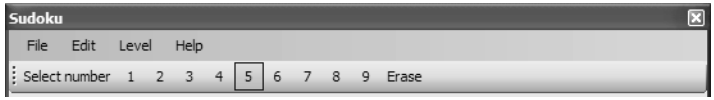


Figure 2-17. *Selecting a number in the toolbar*

Handling Click Events on the Label Controls

When the user has selected a number in the toolbar and clicks a cell on the grid, the `Cell_Click()` event is fired. If a cell already contains a fixed value that was part of the original puzzle (as indicated by a `Tag` property value of `0`, which is not erasable), then there is no need to go further. If the `Tag` property value is `1`, you need to determine the cell that was clicked (through converting the `Sender` object into a Label control and identifying its `Name` property) and then assign it the appropriate value. You will also push the move into a stack data structure so that the user can undo the move later on. Lastly, you need to also check if the puzzle is solved after the value is placed. All these will be serviced by the `Cell_Click()` event, which is coded as follows:

```
'================================================
' Click event for the Label (cell) controls
'================================================
Private Sub Cell_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs)
```

```
'---check to see if game has even started or not---
If Not GameStarted Then
    DisplayActivity("Click File->New to start a new" & _
    " game or File->Open to load an existing game", True)
    Return
End If

Dim cellLabel As Label = CType(sender, Label)

'---if cell is not erasable then exit---
If cellLabel.Tag.ToString() = "0" Then
    DisplayActivity("Selected cell is not empty", False)
    Return
End If

'---determine the col and row of the selected cell---
Dim col As Integer = cellLabel.Name.Substring(0, 1)
Dim row As Integer = cellLabel.Name.ToString().Substring(1, 1)

'---If erasing a cell---
If SelectedNumber = 0 Then
    '---if cell is empty then no need to erase---
    If actual(col, row) = 0 Then Return

    '---save the value in the array---
    SetCell(col, row, SelectedNumber, 1)
    DisplayActivity("Number erased at (" & _
    col & "," & row & ")", False)

ElseIf cellLabel.Text = String.Empty Then
    '---else set a value; check if move is valid---
    If Not IsMoveValid(col, row, SelectedNumber) Then
        DisplayActivity("Invalid move at (" & col & _
        "," & row & ")", False)
        Return
    End If
```

```
            '---save the value in the array---
            SetCell(col, row, SelectedNumber, 1)
            DisplayActivity("Number placed at (" & col & _
            "," & row & ")", False)

            '---saves the move into the stack---
            Moves.Push(cellLabel.Name.ToString() _
            & SelectedNumber)

            '---check if the puzzle is solved---
            If IsPuzzleSolved() Then
                Timer1.Enabled = False
                Beep()
                ToolStripStatusLabel1.Text = "*****Puzzle Solved*****"
            End If

        End If
    End Sub
```

If the puzzle is solved, a beep will sound and a message will be displayed at the bottom of the screen.

Checking Whether a Move Is Valid

Before a value can be assigned to a cell, you must ensure that the value does not violate the rules of Sudoku. That is, it must be the unique number in its column, row, and minigrid. Figure 2-18 shows the checking that must be performed before a cell can be assigned a value. The square indicates the position to insert the value and the shaded regions indicate the cells to check to ensure that the number is unique in its column, row, and minigrid.

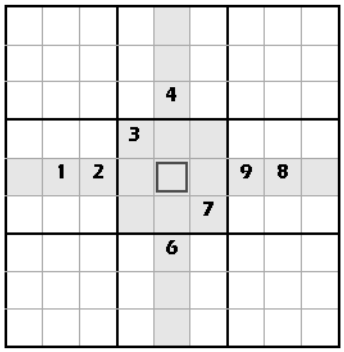


Figure 2-18. *Checking whether a value placed in a location violates the rules of Sudoku*

The IsMoveValid() function checks if a number is valid:

```
'================================================
' Check if move is valid
'================================================
Public Function IsMoveValid( _
   ByVal col As Integer, _
   ByVal row As Integer, _
   ByVal value As Integer) As Boolean

    Dim puzzleSolved As Boolean = True

    '---scan through column
    For r As Integer = 1 To 9
        If actual(col, r) = value Then '---duplicate---
            Return False
        End If
    Next

    '---scan through row
    For c As Integer = 1 To 9
        If actual(c, row) = value Then '---duplicate---
            Return False
        End If
    Next

    '---scan through minigrid
    Dim startC, startR As Integer
    startC = col - ((col - 1) Mod 3)
    startR = row - ((row - 1) Mod 3)

    For rr As Integer = 0 To 2
        For cc As Integer = 0 To 2
            If actual(startC + cc, startR + rr) = value Then
                '---duplicate---
                Return False
            End If
        Next
    Next
    Return True
End Function
```

The IsMoveValid() function first scans the nine columns to see if the number to be inserted has already been used. It then proceeds to scan the nine rows, and finally the nine mingrids. At any point in the scan, if a duplicate is detected, the move is deemed to be invalid and the function returns a False.

Checking Whether a Puzzle Is Solved

After a value is assigned to a cell, you need to check if the puzzle is now solved. The IsPuzzleSolved() subroutine checks the entire grid to determine if the puzzle is solved:

```
Public Function IsPuzzleSolved() As Boolean
    '---check row by row---
    Dim pattern As String
    Dim r, c As Integer
    For r = 1 To 9
        pattern = "123456789"
        For c = 1 To 9
            pattern = pattern.Replace(actual(c, r).ToString(),String.Empty)
        Next
        If pattern.Length > 0 Then
            Return False
        End If
    Next

    '---check col by col---
    For c = 1 To 9
        pattern = "123456789"
        For r = 1 To 9
            pattern = pattern.Replace(actual(c, r).ToString(),String.Empty)
        Next
        If pattern.Length > 0 Then
            Return False
        End If
    Next

    '---check by minigrid---
    For c = 1 To 9 Step 3
        pattern = "123456789"
        For r = 1 To 9 Step 3
            For cc As Integer = 0 To 2
                For rr As Integer = 0 To 2
                    pattern = pattern.Replace( _
                       actual(c + cc, r + rr).ToString(), String.Empty)
```

```
                Next
            Next
        Next
        If pattern.Length > 0 Then
            Return False
        End If
    Next
    Return True
End Function
```

The `IsPuzzledSolved()` function performs checks on the rows, columns, and minigrids. As long as any one of the rows, columns, or minigrids does not have all the numbers from 1 to 9, the subroutine returns a `False`.

Updating the Value of a Cell

The `SetCell()` subroutine assigns a value to a cell by specifying its column and row number, the value to set, and whether it is erasable. Because the cells are represented by Label controls generated dynamically, you need to locate a specific cell by using the `Find()` method in the `Controls` class. The `SetCell()` subroutine also sets the cells using the appropriate colors. Code the `SetCell()` subroutine as follows:

```
'================================================
' Set a cell to a given value
'================================================
Public Sub SetCell( _
   ByVal col As Integer, ByVal row As Integer, _
   ByVal value As Integer, ByVal erasable As Short)

    '---Locate the particular Label control---
    Dim lbl() As Control = _
    Me.Controls.Find(col.ToString() & row.ToString(), True)
    Dim cellLabel As Label = CType(lbl(0), Label)

    '---save the value in the array---
    actual(col, row) = value
    '---set the appearance for the Label control---
    If value = 0 Then '---erasing the cell---
        cellLabel.Text = String.Empty
        cellLabel.Tag = erasable
        cellLabel.BackColor = DEFAULT_BACKCOLOR
```

```
        Else
            If erasable = 0 Then '---means default puzzle values---
                cellLabel.BackColor = FIXED_BACKCOLOR
                cellLabel.ForeColor = FIXED_FORECOLOR
            Else '---means user-set value---
                cellLabel.BackColor = USER_BACKCOLOR
                cellLabel.ForeColor = USER_FORECOLOR
            End If
            cellLabel.Text = value
            cellLabel.Tag = erasable
        End If
    End Sub
```

Figure 2-19 shows the different color coding used to represent different types of values. The lighter shade indicates values set by the user, while the darker shade represents cells set in the original puzzle.

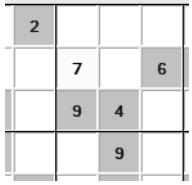


Figure 2-19. *Setting values in the cells*

The `DisplayActivity()` subroutine displays a message in the TextBox control. It also accepts an additional parameter indicating if a beep should be sounded. This is useful for displaying error messages to alert the user. Code the `DisplayActivity()` subroutine as follows:

```
'================================================
' Displays a message in the Activities text box
'================================================
Public Sub DisplayActivity( _
   ByVal str As String, _
   ByVal soundBeep As Boolean)
    If soundBeep Then Beep()
    txtActivities.Text &= str & & Environment.NewLine
End Sub
```

Figure 2-20 shows some messages displayed in the Activities TextBox control.

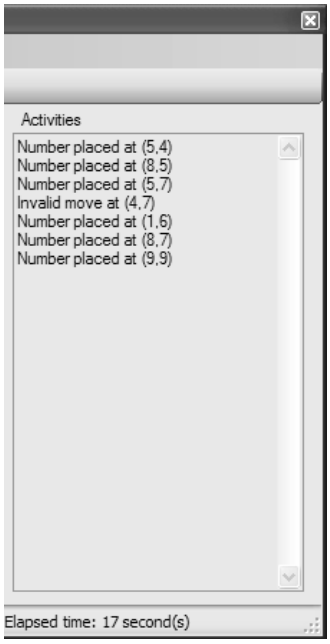


Figure 2-20. *Displaying messages in the TextBox control*

Undoing and Redoing a Move

The user can undo a move by selecting Edit ➤ Undo. To undo a move, you simply need to pop an item from the `Moves` stack and then push it into the `RedoMoves` stack. That way, if the user chooses to redo his move, you can retrieve it from the `RedoMoves` stack as shown in the following event handler for the Undo menu item:

```
'================================================
' Undo a move
'================================================
Private Sub UndoToolStripMenuItem_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles UndoToolStripMenuItem.Click

    '---if no previous moves, then exit---
    If Moves.Count = 0 Then Return

    '---remove from the Moves stack and push into
    ' the RedoMoves stack---
    Dim str As String = Moves.Pop()
    RedoMoves.Push(str)
```

```
        '---save the value in the array---
        SetCell(Integer.Parse(str(0)), Integer.Parse(str(1)), 0, 1)
        DisplayActivity("Value removed at (" & _
           Integer.Parse(str(0)) & "," & _
           Integer.Parse(str(1)) & ")", False)
    End Sub
```

To redo a move, a user selects Edit ➤ Redo. This is similar to undoing a move—instead of popping from the `Moves` stack, you now pop an item from the `RedoMoves` stack and push it into the `Moves` stack. The following event handler for the Redo menu item shows how to redo a move:

```
    '================================================
    ' Redo the move
    '================================================
    Private Sub RedoToolStripMenuItem_Click( _
       ByVal sender As System.Object, _
       ByVal e As System.EventArgs) _
       Handles RedoToolStripMenuItem.Click

        '---if RedoMove stack is empty, then exit---
        If RedoMoves.Count = 0 Then Return

        '---remove from the RedoMoves stack and push into the
        ' Moves stack---
        Dim str As String = RedoMoves.Pop()
        Moves.Push(str)

        '---save the value in the array---
        SetCell(Integer.Parse(str(0)), Integer.Parse(str(1)), _
                Integer.Parse(str(2)), 1)
        DisplayActivity("Value reinserted at (" & _
           Integer.Parse(str(0)) & "," & _
           Integer.Parse(str(1)) & ")", False)
    End Sub
```

Saving a Game

Saving a Sudoku puzzle is surprisingly easy. You can save a Sudoku puzzle as a string of digits. For example, the puzzle shown at the beginning of the chapter in Figure 2-1 is saved in a plain text file containing the following string:

```
402000008
000006240
810940007

080697031
006050900
750020080

600213759
003400006
970000400
```

■ **Note** I have formatted the string in groups of nine for easy reading. In actual fact, this series of digits is saved in the text file as a one-line string.

The SaveGameToDisk() subroutine first determines if the game has already been saved previously. If it has not been saved before (or if the user selects File ➤ Save As), the Save File dialog box is displayed to allow the user to choose a filename. If the file selected already exists, the SaveGameToDisk() subroutine will delete the file and then create a new one to save the string of digits. Code the SaveGameToDisk() subroutine as follows:

```
'================================================
' Save the game to disk
'================================================
Public Sub SaveGameToDisk(ByVal saveAs As Boolean)
    '---if saveFileName is empty, means game has not been saved
    ' before---
    If saveFileName = String.Empty OrElse saveAs Then
        Dim saveFileDialog1 As New SaveFileDialog()
        saveFileDialog1.Filter = _
           "SDO files (*.sdo)|*.sdo|All files (*.*)|*.*"
        saveFileDialog1.FilterIndex = 1
        saveFileDialog1.RestoreDirectory = False
        If saveFileDialog1.ShowDialog() = _
           Windows.Forms.DialogResult.OK Then
            '---store the filename first---
            saveFileName = saveFileDialog1.FileName
```

```
            Else
                Return
            End If
        End If

        '---formulate the string representing the values to store---
        Dim str As New System.Text.StringBuilder()
        For row As Integer = 1 To 9
            For col As Integer = 1 To 9
                str.Append(actual(col, row).ToString())
            Next
        Next

        '---save the values to file---
        Try
            Dim fileExists As Boolean
            fileExists = _
               My.Computer.FileSystem.FileExists(saveFileName)
            If fileExists Then _
               My.Computer.FileSystem.DeleteFile(saveFileName)
            My.Computer.FileSystem.WriteAllText(saveFileName, _
               str.ToString(), True)
            ToolStripStatusLabel1.Text = "Puzzle saved in " & _
                                         saveFileName
        Catch ex As Exception
            MsgBox("Error saving game. Please try again.")
        End Try
    End Sub
```

■ **Note** Realize that I used the `StringBuilder` class for string operation. When manipulating strings in a loop (especially for string concatenation), it is always much more efficient to use a `StringBuilder` class than to append `String` objects directly. Also, the `My` namespace is a new feature in Visual Basic 2005. It is used as a shortcut to the many methods nested deep within the .NET Framework class library.

To save a game, the user can choose File ➤ Save As. The following shows the event handler for the Save As menu item:

```
'================================================
' Save as... menu item
'================================================
Private Sub SaveAsToolStripMenuItem_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles SaveAsToolStripMenuItem.Click

    If Not GameStarted Then
        DisplayActivity("Game not started yet.", True)
        Return
    End If

    SaveGameToDisk(True)
End Sub
```

If a game has previously been saved, the user can just choose File ➤ Save. The following shows the event handler for the Save menu item:

```
'================================================
' Save menu item
'================================================
Private Sub SaveToolStripMenuItem_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles SaveToolStripMenuItem.Click
    If Not GameStarted Then
        DisplayActivity("Game not started yet.", True)
        Return
    End If
    SaveGameToDisk(False)
End Sub
```

Opening a Saved Game

To open a previously saved game from disk, you first ask the user if she wants to save the current game. You then invoke the `StartNewGame()` subroutine and prompt the user to specify the filename of the saved game. You then initialize the individual cells of the grid based on the content of the file opened. The following shows the event handler for the Open menu item:

```
'================================================
' Open a saved game
'================================================
Private Sub OpenToolStripMenuItem_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles OpenToolStripMenuItem.Click

    If GameStarted Then
        Dim response As MsgBoxResult = _
        MessageBox.Show("Do you want to save current game?", _
                        "Save current game", _
                        MessageBoxButtons.YesNoCancel, _
                        MessageBoxIcon.Question)

        If response = MsgBoxResult.Yes Then
            SaveGameToDisk(False)
        ElseIf response = MsgBoxResult.Cancel Then
            Return
        End If
    End If

    '---load the game from disk---
    Dim fileContents As String
    Dim openFileDialog1 As New OpenFileDialog()
    openFileDialog1.Filter = _
       "SDO files (*.sdo)|*.sdo|All files (*.*)|*.*"
    openFileDialog1.FilterIndex = 1
    openFileDialog1.RestoreDirectory = False

    If openFileDialog1.ShowDialog() = _
       Windows.Forms.DialogResult.OK Then
        fileContents = _
           My.Computer.FileSystem.ReadAllText( _
              openFileDialog1.FileName)
        ToolStripStatusLabel1.Text = openFileDialog1.FileName
        saveFileName = openFileDialog1.FileName
    Else
        Return
    End If
```

```
    StartNewGame()

    '---initialize the board---
    Dim counter As Short = 0
    For row As Integer = 1 To 9
        For col As Integer = 1 To 9
            Try
                If CInt(fileContents(counter).ToString()) <> 0 Then
                    SetCell(col, row, _
                       CInt(fileContents(counter).ToString()), 0)
                End If
            Catch ex As Exception
                MsgBox( _
                   "File does not contain a valid Sudoku puzzle")
                Exit Sub
            End Try
            counter += 1
        Next
    Next
End Sub
```

Ending the Game

To end the game, the user simply chooses File ➤ Exit. Before exiting the application, prompt the user to save the game. The following shows the event handler for the Exit menu item:

```
'================================================
' Exit the application
'================================================
Private Sub ExitToolStripMenuItem_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles ExitToolStripMenuItem.Click
    If GameStarted Then
        Dim response As MsgBoxResult = _
        MsgBox("Do you want to save current game?", _
        MsgBoxStyle.YesNoCancel, "Save current game")
```

```
            If response = MsgBoxResult.Yes Then
                SaveGameToDisk(False)
            ElseIf response = MsgBoxResult.Cancel Then
                Return
            End If
        End If
        '---exit the application---
        End
    End Sub
```

Testing the Application

Now that the application is all wired up, it is time to test the application. In Visual Studio 2005, press F5 to debug the application.

Save the following in a text file and save it as `C:\Easy.sdo`:

```
005400180146080500070013000451008706080000010603700948000390070004070269019006400
```

In the Sudoku application, load the `Easy.sdo` file by choosing File ➤ Open and selecting `C:\Easy.sdo`. The Sudoku puzzle should now look like Figure 2-21.

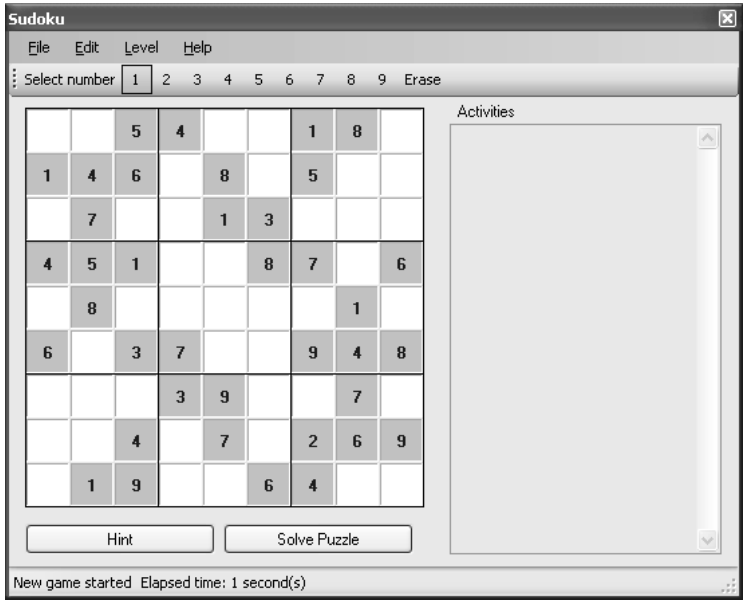


Figure 2-21. *Loading a Sudoku puzzle*

Try solving the puzzle and see how long it takes.

■ **Tip** This is an easy Sudoku puzzle.

Give up? Figure 2-22 shows the solution for the puzzle!

3	9	5	4	6	7	1	8	2
1	4	6	2	8	9	5	3	7
2	7	8	5	1	3	6	9	4
4	5	1	9	3	8	7	2	6
9	8	7	6	4	2	3	1	5
6	2	3	7	5	1	9	4	8
5	6	2	3	9	4	8	7	1
8	3	4	1	7	5	2	6	9
7	1	9	8	2	6	4	5	3

Figure 2-22. *The solution to the Sudoku puzzle*

Summary

In this chapter, you have walked through the various steps to construct a Sudoku puzzle board using a Windows application. This is the foundation chapter that all future chapters will build on. Although the application in this chapter lacks the intelligence required to solve a Sudoku puzzle, it does allow you to play Sudoku on the computer. Moreover, the application that you built in this chapter provides some aid to beginning Sudoku players because it checks for compliance with the rules of Sudoku. Go find a Sudoku puzzle and load it using this application. You will gain a better appreciation of the game after a few rounds.

In the next chapter, you are going to discover the first steps toward programmatically solving a Sudoku puzzle. You will be surprised to learn that a lot of Sudoku puzzles can actually be solved by using the simple logic detailed in Chapter 3.