

Responding to the Keyboard in .NET

The first point to make about the keyboard in Windows is that quite possibly you can write your program *without* writing any code yourself to deal with the keyboard. You can often handle user keyboard input through the *TextBox* or *RichEdit* controls; and the customary uses of the tab key for navigating in a dialog, and the *Enter* and *Esc* (escape) keys for providing keyboard equivalents of *OK* and *Cancel* are built into the .NET framework, so no programming is required. The same goes for shortcut keys on menus.

However, sometimes you will want to respond to key presses, and this is the lecture in which you will learn how to do it.

Focus

The first concept connected with the keyboard is *the focus*. At any given moment, one and only one window has “the focus”, or “the keyboard focus”. (There is only one kind of focus, so it is safe to omit the word “keyboard” when it’s clear.) That is the window that will receive keyboard messages until the focus changes. This is a fundamental feature of Windows, not just a .NET concept. Therefore, before even beginning to study the keyboard itself or the different kinds of keyboard events, you should understand the rules governing which window will have the focus. Some of these rules arise from Windows itself, and others are .NET specific. Failing to know or obey these rules results in programmer frustration, when the window you think should get keyboard events does not, and your program unfortunately remains unresponsive to the keyboard.

In .NET programming, a window that can have the focus must be represented by a .NET class derived from *Control*. Note that *Form* is derived from *Control*, so any form meets this condition. In order for a control to (be able to) have the focus, all of the following must be true, according to the documentation of *Focus*.

- Its *Visible* and *Enabled* properties must be *true*.
- It must be contained in another control.
- All of its parent controls must also be *Visible* and *Enabled*. (This means that the parents of its parents, etc.)

- Its *ControlStyles.Selectable* property must be *true*.

For example, the following standard controls are *not* selectable, so they never get the keyboard focus: *Panel*, *GroupBox*, *Splitter*, *Label*, *ProgressBar*, *PictureBox*, and a *LinkLabel*, when it doesn't contain a link.

Notice that a top-level form such as *Form1* does not meet the criterion that it must be contained in another control. Nevertheless, and in spite of the Microsoft documentation just quoted, it *can* receive focus and respond to keyboard events, **but only if it does not contain any other controls**. Thus, you can write a nice keyboard-responsive program, and then add just one button to the form, and suddenly it will no longer respond to the keyboard! Comment out the single line of code that adds the button, and the keyboard comes alive again. You might not like that situation, but it's your fault: Microsoft *told* you that the keyboard would only work if the control is contained in another control. If they allowed it to work anyway under some circumstances, that was just a free gift—you had no right to expect it.

The moral of this story: Design your program so that the parts of it that need to respond to the keyboard are isolated in custom-designed controls. Add one of those controls to the form when you need one.

Key Presses and Characters

We have looked at the computer end of the keyboard interface; now it's time to look at the keyboard itself. Although you use the keyboard every day, you may not have thought about it—that's a sign that it is well-designed. When things work well, we don't think about them. The keyboard consists of the keys that you can press, plus some electronic circuitry that translates your physical key presses into signals that are sent to the computer (over a cable, or nowadays sometimes over a wireless connection). These signals are of two kinds:

- bytes that tell which physical key was pressed
- bytes that specify a character code

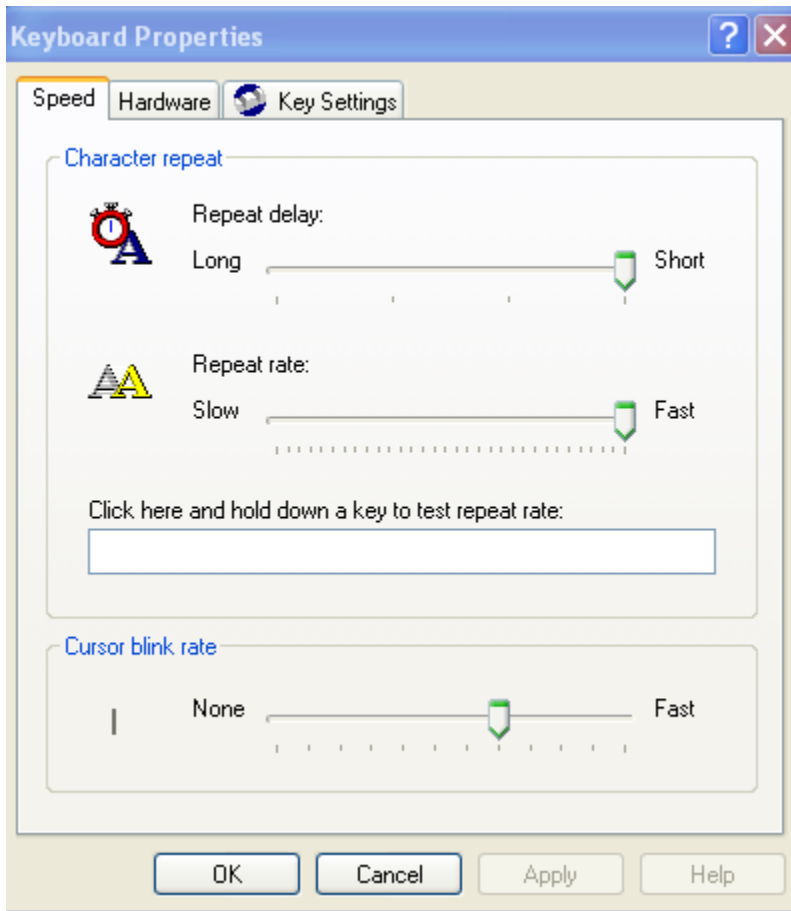
For example, there are usually two different keys you can press to send a numeral 7—one on the numeric keypad, and one on the row of keys above the usual letters of the alphabet. The following classification of the keys on a keyboard may be helpful:

- Character keys. These produce a character code byte as well as a byte identifying the key.
- Toggle keys: *NumLock*, *CapsLock*, and *ScrollLock*, and in some situations *Ins*. These keys change the “state” of the other keys.
- Shift keys: *Shift*, *Crtl*, *Alt*. These keys also affect the interpretation of the other keys, but only while they are held down. Thus the key combination of pressing *Shift* and the “A” key will produce character code 65 (or ‘A’), while pressing the “A” key alone will produce character code 97, or ‘a’.
- Non-character keys: for example, the function keys and the arrow keys. These produce only a key-identification byte (no character code) and do not alter the operation of the other keys.

When does the keyboard send these bytes of information to the computer?

- When a key is depressed
- When a key is released
- When the *auto repeat* feature is triggered.

Auto repeat causes multiple character codes to be sent by character keys when the key is held down for “a long time”. The lengths of time before auto repeat kicks in, and the interval between auto repeats, are both configurable by the user of the keyboard. All modern operating systems provide some interface for the user to do this. For example:



Connection between the keyboard and Windows

The distinction between these kinds of keys and what they send to the computer is independent of any operating system. This depends on the keyboard hardware. The BIOS receives these bytes and generates a “keyboard interrupt”. Control is transferred to a certain location. When Windows is running, the code that is found there was installed by Windows. That code constructs certain Windows messages from the information supplied by the BIOS, and places those messages in the application message queue of the window that currently has the focus.

Keyboard Layouts

The connection between the character code and the key code is determined by the “keyboard layout”. The keyboard layout varies from country to country. Keyboards in Mexico or Italy are hard for Americans to type on, because “the letters are in different places”. Even on your standard

American English Windows, you can install other keyboard layouts from *Control Panel*. You can then easily switch between the installed keyboard layouts—I do this to type Spanish, by choosing the *US International* keyboard layout. Selecting a different keyboard layout in Windows results in different character codes being placed in the messages constructed by the Windows keyboard interrupt handler. This affects all Windows programs, of course. Keyboard layouts are a feature of the operating system, not of the keyboard itself. The character codes generated by the keyboard are always the same for a given keyboard, but Windows “translates” them according to the keyboard layout.

Keyboard Events *KeyDown* and *KeyUp*

The *KeyUp* and *KeyDown* event handlers get an argument from which you can extract the key code, i.e., which actual key was pressed, as opposed to what character. The *KeyPress* event is more convenient if you only care what character was typed. For example, you could not use a *KeyPress* event handler to tell whether “7” was typed on the numeric keypad or from the key above “Y” and “U”; or to tell when function key F7 has been pressed; but for use in, say, a crossword puzzle program, it would suffice.

The argument of the event handler has a *Keys* property, whose value identifies the key. Its value is an enumeration type; the values are things like *Keys.A*, *Keys.B*, *Keys.F7*. These values identify keys regardless of their shift state; that is, you get *Keys.A* whether or not the *Shift* or *Ctrl* key or both were also down while the A key was typed. The keys on the row above the letter keys are *Keys.D0*, *Keys.D1*, *Keys.D2*, etc. The “7” key on the numeric keypad is *Keys.NumPad7*. But when *NumLock* is toggled off, you *do* get different key codes for the numeric keypad keys, for example *Keys.Home*. In other words, the key codes are independent of the shift state, but not of the states toggled by *NumLock*. On the other hand, upper case and lower case letters have the same key codes anyway, so *CapsLock* doesn’t affect key codes.

The complete list of values in the *Keys* enumeration can be found in the documentation. There are values for up to 24 function keys and assorted other keys that are only found on some keyboards.

Detecting Shift States in *KeyDown* and *KeyUp*

If *e* is the argument passed to your *KeyDown* or *KeyUp* handler, then *e.KeyData* is a 32-bit integer. The lower 16 bits are used for the key code, and the upper 16 bits are used to tell whether *Control*, *Shift*, or *Alt* are depressed. Thus if you test

```
if(e.KeyData == Keys.A)
```

you will not catch *Shift-Alt-A*. To catch *Shift-Alt-A* you would need to test

```
if(e.KeyData == Keys.Alt | Keys.Shift | Keys.A).
```

Note that you use a bitwise “or” to express “Alt is depressed *and* A is depressed”. This often confuses students, so let’s look at it in detail. *Keys.Alt* is actually 0x00040000, which in binary is

```
00000000000001000000000000000000
```

Keys.A is 65, which in binary is

```
00000000000000000000000001000001
```

Their bitwise or, which is what you are testing for, is

```
00000000000001000000000001000001.
```

That is the *KeyData* you will get when *Alt-A* is pressed but the *Shift* key is not pressed. Testing for this key data would detect *Alt-a*. If you want to test for *Shift-Alt-A*, you need to also take the bitwise or with *Keys.Shift*, which is

```
00000000000000010000000000000000
```

in binary. Taking the bitwise or of this with the previous value yields

```
00000000000001010000000001000001.
```

This is the value of *Keys.Alt | Keys.Shift | Keys.A*, the key data you would get when *Shift-Alt-A* is pressed.

The most likely case in which you would need to know this is that you want to make the function keys do something useful in your program. Usually *Alt* is used to make shortcut keys for menu items, and .NET offers you direct

support for that, without programming the keyboard yourself, although now you know how the .NET programmers did it.

There are two additional useful fields in the *KeyEventArgs* type. If *e* is the argument to your *KeyDown* or *KeyUp* handler, then *e.KeyCode* is of type *Keys* and represents the lower 16 bits of *KeyData*. In other words, it tells you what key was pressed, without the shift-state information. You could cast that value to an integer, but to save you the trouble, you can just use *e.KeyValue*, which is of type *Int32* (the FCL version of C#'s *int*) and is just *e.KeyCode* cast to an *int*.

The KeyPress Event

If all you care about is the character that was typed, then you don't need to process *KeyUp* and *KeyDown*. It will suffice to process *KeyPress*. The argument passed to the handler of this event has a *KeyChar* property that gives you the character code directly. Simple keyboard processing may use only this event.

CapsLock and NumLock: Calling the Win32 API

These keys have a *state*, which is either *off* or *on*. Suppose you wanted to respond differently to the 'A' key depending on whether *CapsLock* is *on* or *off*. Guess what: this can't be done in .NET, apparently. There is no way to determine, using the FCL, whether *CapsLock* is on or off! You can keep track of how many times it has been pressed, but that won't help you if you don't know whether it was on or off when your program started. (It really can't be done-- it's not simply that I didn't discover how to do it--see the statement of Petzold, p. 228 of his book *Programming Windows in C#*). However, it *can* be done in the Win32 API, so this is one of those occasions when if you want to accomplish the task, you will have to use the Win32 API. The method for doing this is the same as to call any function defined in some DLL (dynamic link library). Here is how:

First, create a new class to hold the function or functions you want to call. For that matter, you could do this within an existing class, but it seems cleaner to create a class, perhaps called *Win32*, to hold any Win32 functions you may need. At the top of your new class file you will need the following directives:

```
using System;
using System.Runtime.InteropServices;
```

Then you can put a class definition containing the external function you want to call. You can leave it inside the namespace brackets that Visual Studio wrote for you, or you can delete those namespace brackets, in which case the file will be easier to re-use in another project later. The mysterious code in square brackets is an “attribute” of the function declaration that follows. It tells (at the minimum) what DLL to search for the entry point of the function. The function itself must be declared *static extern*. It could be *public* or *private*; here I’ve chosen *public*.

```
public class Win32
{
    public Win32()
    {}
    [DllImport("user32.dll",
              CharSet=CharSet.Auto,
              ExactSpelling=true,
              CallingConvention=CallingConvention.Winapi)]
    public static extern short GetKeyState(int keyCode);

    public bool GetCapsLock()
    { bool CapsLock =
      (((ushort) GetKeyState(0x14)) & 0xffff) != 0;
      /* 0x14 is VK_CAPITAL in the Win32 header files*/
      return CapsLock;
    }
    public bool GetNumLock()
    { bool NumLock =
      (((ushort) GetKeyState(0x90)) & 0xffff) != 0;
      /* 0x90 is VK_NUMLOCK */
      return NumLock;
    }
}
```

Tab and Arrow Keys

If you don’t do something special to prevent it, you won’t be able to respond to the tab and arrow keys, because your handler for *KeyDown* or *KeyPress* will never receive the event! Here is what the documentation for the *KeyDown* event says:

Certain keys, such as the TAB, RETURN, ESCAPE, and arrow keys are handled by controls automatically. In order to have these keys raise the *KeyDown* event, you must override the *IsInputKey* method in each control on your form. The code for the override of the *IsInputKey* would need to determine if one of the special keys is pressed and return a value of **true**.

In order to make sure that you get *all* keyboard events when your form (or control) has the focus, you can use this code:

```
protected override bool IsInputKey( Keys keyData )
{
    return true;
}
```

On the other hand, you may want to handle all the keyboard events occurring in any of the controls on a form in the form itself, rather than in the individual controls. Here is what the documentation says about *that*:

To handle keyboard events only at the form level and not allow other controls to receive keyboard events, set the [KeyPressEventArgs.Handled](#) property in your form's **KeyPress** event-handling method to **true**.

These two pieces of information show us that keyboard events go first to the top-level form (one with no parent) owning the window with the focus. There they are pre-processed by *IsInputKey* and, if it returns *false*, then they are processed by the default keyboard event handlers of the top-level form. But if it returns *true* (as it will if has been overridden as shown), then the keyboard event is processed by the form, and then, unless the *Handled* property of the event is set to true, is passed on down to the child window with the focus; or to the parent or grandparent of the window with the focus if the window with the focus is a granddaughter or great-granddaughter window of the form rather than a child window.

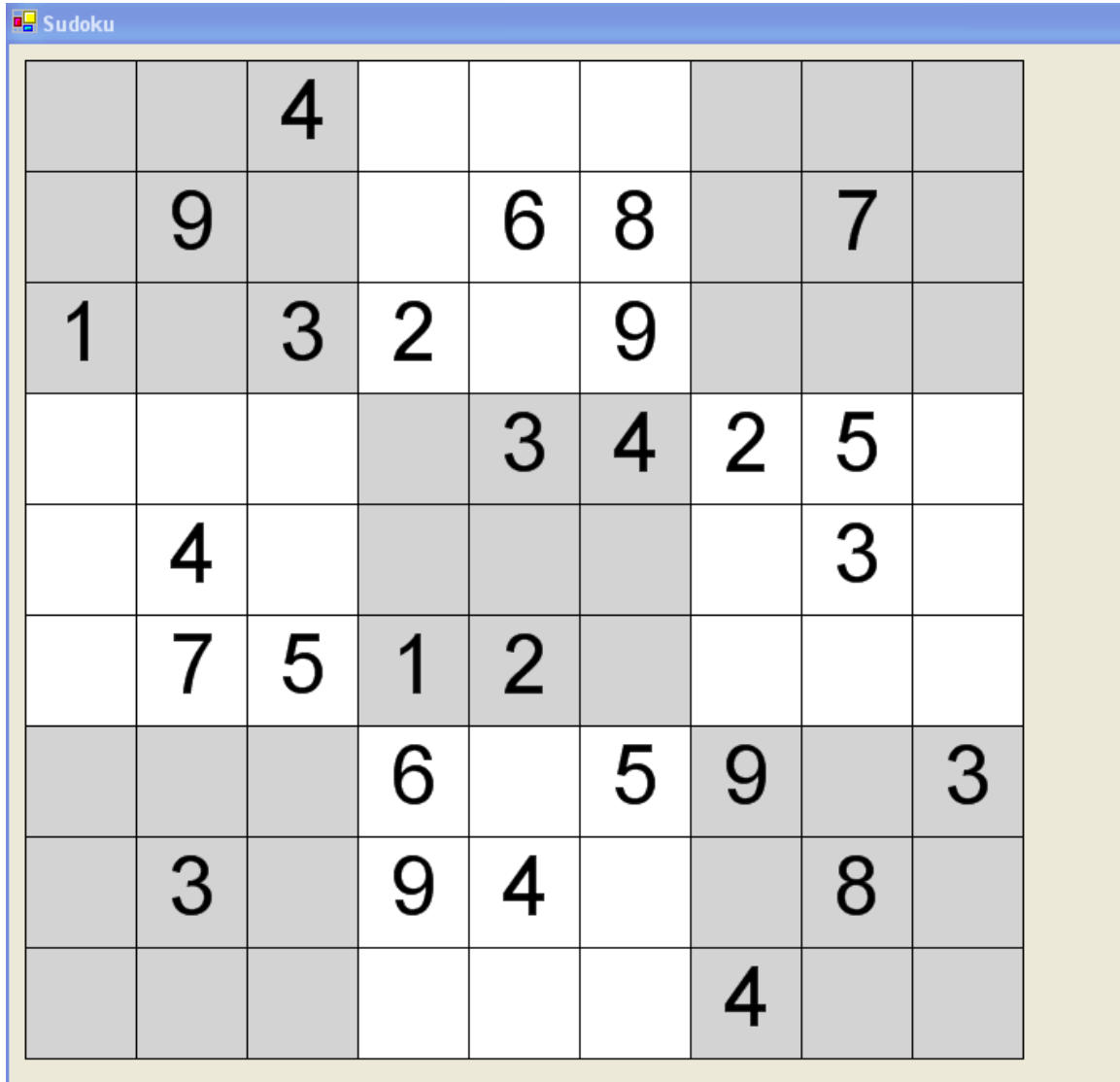
Petzold's program *KeyExamine* (pp. 233 ff. of his book *Programming Windows in C#*, which is a reference book but not the textbook for this course) is in a *Form* and does not have this problem. But if he were to add another control to his form, say a single button, he would need to override *IsInputKey* as shown here.

Using *KeyDown* in data validation

When we studied data validation, we did not consider this possibility: if the user is required to enter a non-negative integer in a certain text box, maybe we could simply have that textbox refuse to accept any characters but digits. First note that, even if we do that, we probably will still have to validate the data, since the integer might be too large to store in an *int*, or more likely, the program will have further constraints on the possible value. But still, it is possible to prevent non-digit entry (or enforce other constraints on text entry) by installing a *KeyDown* handler for the textbox. To discard a keypress, set *e.Handled* to *true*, where *e* is the *KeyEventArgs* parameter passed to the handler.

Sudoku

As an example program, we will take up the popular Japanese puzzles known as *Sudoku*. Here is a sample of such a puzzle.



To solve the puzzle, you enter digits into the empty squares in such a way that

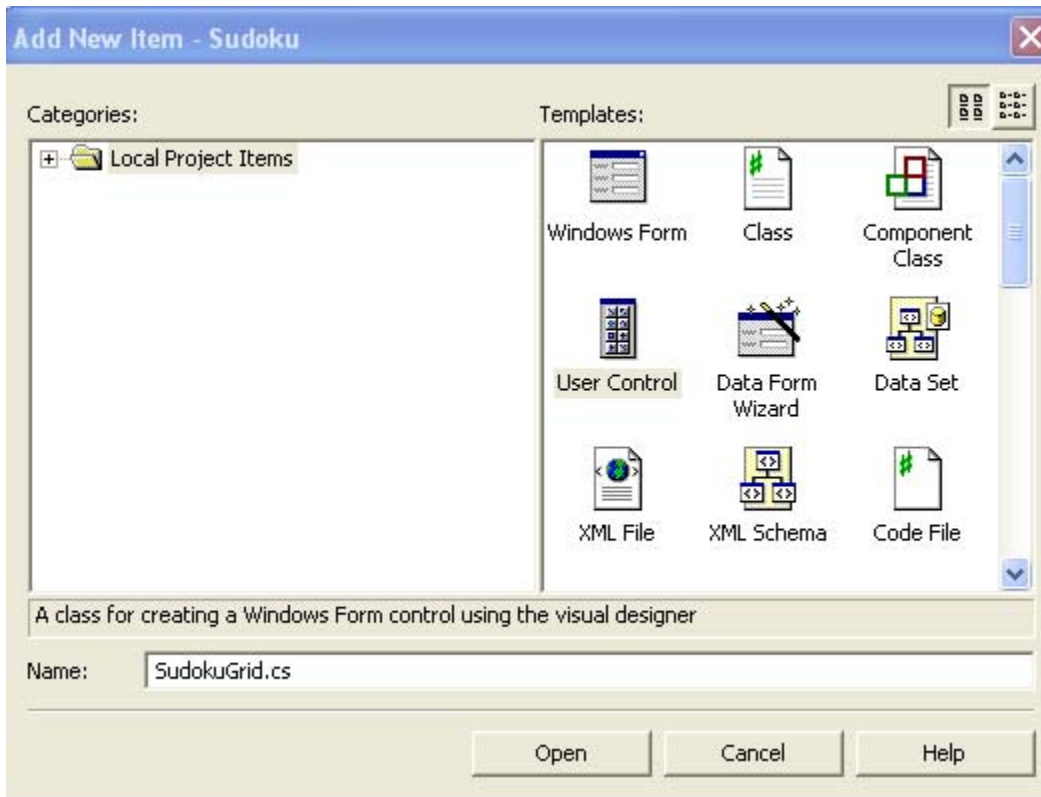
- Every row contains each of 1,2,...,9 exactly once
- Every column contains each of 1,2,...,9 exactly once.
- Each of the 9 shaded 3 by 3 rectangles contains 1,2,...,9 exactly once.

These puzzles, incidentally, appear daily in the *San Francisco Chronicle*. Tuesday's puzzle is easiest; Wednesday's is harder, and so on through the week. They also appear in the *San Jose Mercury*, but I don't know if the difficulty varies with the day. [I'm not sure if they still do appear as claimed—it was true in 2007 at least.]

Our example program will present a 9 by 9 grid, possibly already containing some numbers, and let the user click on a square and then type a digit. If the digit, together with the numbers already present, does not violate the rules above, then it should appear in the grid.

Start by creating a new application *Sudoku*. Since we later will want to add some other controls to the main form to provide tools to help the user solve the puzzle, we will create the *Sudoku* grid itself in a separate control. Otherwise, it will work fine until we add a single button, and then it won't get the focus any more and will stop working. (Guess how I discovered this... ☹). That was before I knew about *IsInputKey*. I think we could create *Sudoku* all in *Form1* if we override *IsInputKey* to always return *true*. But anyway, it's good object-oriented programming to isolate the display of the Sudoku grid in a separate class.

To do this, we choose *Project / Add Windows Form* in Visual Studio, and choose *User Control* as shown here:



The result of choosing *User Control* instead of *Windows Form* is that the first line of code comes out like this:

```
public class SudokuGrid : System.Windows.Forms.UserControl
```

instead of like this:

```
public class SudokuGrid : System.Windows.Forms.Form
```

That makes a difference when you add the following code in *Form1.cs*:

```
private SudokuGrid m_theGrid;
public Form1()
{
    InitializeComponent();
    m_theGrid = new SudokuGrid();
    m_theGrid.Location = new Point(10,10);
    this.Controls.Add(m_theGrid);
    ...
}
```

If you used *Windows Form* instead of *User Control*, the *Controls.Add* command will generate a run-time error. You are not allowed to add a top-level form to the controls list.

Now, all the work of drawing the grid and processing mouse and keyboard messages will in the *SudokuGrid* class. Here is how the program will look to the user. (Since this is a lecture about the keyboard, the interface designed here is primarily keyboard-oriented.)

- You will initially see a blank screen and be able to enter digits to create a puzzle. You can click in a blank square and then type any digit that does not violate the rules of *Sudoku*.
- Digits written with the *Alt* key depressed can be overwritten if and only if *Alt* is depressed at the time of overwriting. Such “permanent” digits appear in black. (They are used for entering the puzzle.)
- Digits written without the *Alt* key depressed can be overwritten with or without *Alt* depressed. (Whether *Alt* is depressed will determine whether the new character can be overwritten later.)
- To overwrite an immutable digit (one written with *Alt* depressed) and replace it by a mutable digit, you must hold down both the *Alt* and *Shift* keys.
- No digit can ever be entered that violates the rules of *Sudoku*.
- If a digit can be overwritten, space bar overwrites it with a blank.
- Arrow keys move the focus one square to the left, right, up, or down. Of course technically the focus is possessed by the *SudokuGrid* control, but I mean which square will show the next digit.
- The square that has the focus will have a visual indication of this fact, by a colored border just inside the square.

Of course, one can think of a lot more features that would be nice in such a program, but this list is probably more than enough for a half-hour demonstration, and covers several different keyboard issues. Originally, I had also listed this feature:

- The tab key moves you to the next of the nine 3 by 3 shaded sub-grids, in reading order. Shift-tab goes the other way.

But if we later want to put any other controls on the form, besides the *SudokuGrid*, we will want the tab key to perform its usual function of moving between controls, so we had better not give it a special function within the *SudokuGrid*.

Before starting to program this example, we should think about the data structures required. We can either have several different 9 by 9 arrays (of digits, rectangles, immutability flag, etc.), or we could create a new class *Entry* with these fields, and have a 9 by 9 array of *Entries*. The latter is the way to go—it's easier to extend, it's easier to write, it's easier to read:

```
public class Entry
{
    public Rectangle r;//square in which the entry is displayed
    public char digit = ' '; // the digit displayed
    public bool focus = false; // keystroke affects this entry?
    public bool mutable = true;
        // overwritable with caps lock off?
    public int i,j; // row and column position in Sudoku

    private static Brush immutableBrush = Brushes.Black;
    private static Brush mutableBrush = Brushes.Blue;
    private static Pen focusPen = Pens.Yellow;
    private static Font smallfont = new Font("Arial", 8);
    private static Font bigfont = new Font("Arial", 40);
    private static StringFormat fmt = new StringFormat();

    public Entry(int I, int J)
    {
        fmt.Alignment = StringAlignment.Center;
        fmt.LineAlignment = StringAlignment.Center;
        i = I;
        j = J;
    }
    public void Draw(Graphics g)
    { Brush b;
      if(((i/3+j/3) & 1) == 1)
          b = Brushes.White;
      else
          b = Brushes.LightGray;
```

```

g.FillRectangle(b,r);
g.DrawRectangle(Pens.Black,r);
if(focus)

    if(digit != ' ')
        { String s = new String(digit,1);
          Brush b = mutable ? mutableBrush : immutableBrush;
          g.DrawString(s,bigfont,b,r,fmt);
        }
}}

```

Now that *Entry* objects can draw themselves, the *Paint* handler in *Form1* is supremely simple:

```

private void SudokuGrid_Paint(object sender,
System.Windows.Forms.PaintEventArgs e)
{
    Graphics g = e.Graphics;
    int i,j;
    for(i=0;i<9;i++)
    for(j=0;j<9;j++)
        m_Grid[i,j].Draw(g);
}

```

```

private void SudokuGrid_MouseDown(object sender,
System.Windows.Forms.MouseEventArgs e)
{
    int i,j;
    for(i=0;i<9;i++)
    {
        for(j=0;j<9;j++)
            {
                if(m_Grid[i,j].r.Contains(e.X,e.Y))
                    {
                        if(m_Focus.X >= 0 && m_Focus.Y >= 0)
                            { m_Grid[m_Focus.Y,m_Focus.X].focus =
                                false;
                                Invalidate(m_Grid[m_Focus.Y,m_Focus.X].r);
                            }
                        m_Focus.Y = i;
                        m_Focus.X = j;
                        m_Grid[i,j].focus = true;
                        Invalidate(m_Grid[i,j].r);
                        break;
                    }
            }
        if(j<9)
            break;
    }
}

```

In order to test this mouse and graphics code, let's put in a primitive *KeyPress* handler:

```
private void SudokuGrid_KeyPress(object sender,
    System.Windows.Forms.KeyPressEventArgs e)
{ if(m_Focus.X < 0 || m_Focus.Y < 0)
    return;
  char c = e.KeyChar;
  if('0' <= c && c <= '9')
    { m_Grid[m_Focus.Y,m_Focus.X].digit = c;
      Invalidate(m_Grid[m_Focus.Y,m_Focus.X].r);
    }
}
```

This allows us to create a puzzle and test the mouse and graphics code, but it barely begins to implement the keyboard interface specified. To do that, we start with the code discussed above, to enable us to capture the *KeyDown* events for the tab and arrow keys.

```
protected override bool IsInputKey( Keys keyData)
{ return true; }
```

Here's an improvement, to implement the check for legality and the proper handling of the *Alt* key to ensure that characters written with the *Alt* key down can only be overwritten with the *Alt* key down. You'll want to remove the *KeyPress* handler above—it was only for an initial test. Everything will happen in *KeyDown*.

```
private bool legal(int key, int i, int j)
// only accept digits and space, and
// don't allow the rules of Sudoku to be violated.
// i and j are the row and column about to be written to.
{ if(c == ' ')
    return true;
  if('0' > key || key > '9')
    return false;
  char c = (char) key;
  int p,u,v;
  for(p=0;p<9;p++)
    { if(m_Grid[p,j].digit == c && p != i)
        return false;
      if(m_Grid[i,p].digit == c && p != j)
        return false;
    }
```

```

    u = i/3;
    v = j/3;
    if(m_Grid[u+p/3,v+p%3].digit == c &&
        !(u+p/3 == i && v+p%3 == j)
        )
        return false;
    }
return true;
}

```

```

private void SudokuGrid_KeyDown(object sender,
System.Windows.Forms.KeyEventArgs e)
{
    if(m_Focus.X < 0 || m_Focus.Y < 0)
        return;
    Keys c = e.KeyCode;
    if(c == Keys.Left ||
        c == Keys.Right ||
        c == Keys.Down ||
        c == Keys.Up)
    { // arrow keys navigate the grid
        int deltax = 0, deltay = 0;
        if(c == Keys.Left)
            { deltax = -1;
              deltay = 0;
            }
        else if (c == Keys.Right)
            { deltax = 1;
              deltay = 0;
            }
        else if(c == Keys.Up)
            { deltax = 0;
              deltay = -1;
            }
        else if(c == Keys.Down)
            { deltax = 0;
              deltay = 1;
            }
        int I = m_Focus.Y + deltay;
        int J = m_Focus.X + deltax;
        if(I > 8) I -= 9; // arrow keys wrap around
        if(I < 0) I += 9;
        if(J > 8) J -= 9;
        if(J < 0) J += 9;
        m_Grid[m_Focus.Y,m_Focus.X].focus = false;
        // leaving that square
        Invalidate(m_Grid[m_Focus.Y,m_Focus.X].r);
    }
}

```

```

        m_Grid[I,J].focus = true;
        Invalidate(m_Grid[I,J].r);
        // going to that square;
        m_Focus.Y = I;
        m_Focus.X = J;
        return;
    }
    int v = e.KeyValue;
    if(legal(v,m_Focus.Y,m_Focus.X) &&
        (m_Grid[m_Focus.Y,m_Focus.X].mutable || e.Alt)
    )
    { m_Grid[m_Focus.Y,m_Focus.X].digit = (char) v;
      Invalidate(m_Grid[m_Focus.Y,m_Focus.X].r);
      if(m_Grid[m_Focus.Y,m_Focus.X].mutable == false)
          { if(e.Alt && e.Shift)
              m_Grid[m_Focus.Y,m_Focus.X].mutable = true;
            else if(e.Alt)
              m_Grid[m_Focus.Y,m_Focus.X].mutable = false;
          }
      else if(e.Alt)
          // making a formerly mutable square immutable
          m_Grid[m_Focus.Y,m_Focus.X].mutable = false;
    }
}

```

The program in this condition implements the keyboard interface described above.