To create a project and add a table to your form

1. On the **File** menu, click **New Project**.
2. If you’re not using Visual Studio Express, you need to select a language first. From the **Installed Templates**list, select either **C#** or **Visual Basic**.
3. Click the **Windows Forms Application** icon, and then type **MatchingGame** as the name.
4. Set the form properties:
   1. Change the form's **Text** property to **Matching Game**.
   2. Change the size to 550 pixels wide by 550 tall by using the **Size** property or dragging until you see the correct size in the lower-left corner of the integrated development environment (IDE).
5. Drag a **TableLayoutPanel** control from the Toolbox, and then set its properties:
   1. Set the **BackColor** property to **CornflowerBlue**. (Select the **Web** tab in the color picker to see the color names.)
   2. Set the **Dock** property to **Fill** by clicking the drop-down button next to the property and clicking the large middle button.
   3. Click the triangle button in the upper-right corner of the TableLayoutPanel to display the task menu.
   4. Click **Add Row** twice to add two more rows, and then click **Add Column** twice to add two more columns.
   5. Click **Edit Rows and Columns** to open the **Column and Row Styles** window. Select each of the columns, click the **Percent** button and set each column's width to 25 percent of the total width. Then select **Rows** from the drop-down box at the top of the window, and set each row's height to 25 percent. Click the **OK** button.

Your TableLayoutPanel should now have sixteen equal-size square cells.

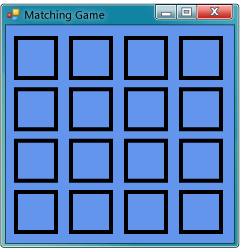
1. Be certain that the TableLayoutPanel is selected in the form editor. While it's selected, open the Toolbox and double-click **Label** to add a **Label** control to the upper-left square. The **Label** control should now be selected in the IDE. Set its properties:
   1. Set the **BackColor** property to **CornflowerBlue**.
   2. Set the **AutoSize** property to **False**.
   3. Set the **Dock** property to **Fill**.
   4. Set the **TextAlign** property to **MiddleCenter** by clicking the drop-down button next to the property, and then clicking the middle button.
   5. Click the **Font** property. An ellipsis button should appear.
   6. Click the ellipsis button, and set the font to **Webdings 72 point Bold**.
   7. Set the **Text** property to the letter **c**.

The upper-left cell in the TableLayoutPanel should now contain a big black box centered on a blue background.

|  |
| --- |
| **NoteNote** |
| The Webdings font is a font of icons that ships with the Microsoft Windows operating system. In your matching game, the player needs to match pairs of icons, so you use this font to display the icons to match. Instead of putting **c** in the **Text** property, try entering different letters to see what icons are displayed. An exclamation point is a spider, an uppercase N is an eyeball, and a comma is a chili pepper. |

1. Select your **Label** control and copy it. (Press Ctrl+C or from the **Edit** menu, click **Copy**.) Then paste it. (Press Ctrl+V or from the **Edit** menu, click **Paste**.) Another label appears in the second cell in the TableLayoutPanel. Paste it again, and another label appears in the third cell. Keep pasting **Label** controls until all of the cells are filled.

|  |
| --- |
| **NoteNote** |
| If you paste too many times, the IDE adds a new row to the TableLayoutPanel so that it has a place to add your new **Label** control. You can undo it. To remove the new cell, press Ctrl+Z or on the **Edit** menu, click**Undo**. |

1. Now your form is laid out. It should look the following picture.
2. Initial matching game form
3. 

To add a Random object and a list of icons

1. Before you add the following code to create the list, consider how it works.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553237(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

public partial class Form1 : Form

{

// Use this Random object to choose random icons for the squares

Random random = new Random();

// Each of these letters is an interesting icon

// in the Webdings font,

// and each icon appears twice in this list

List<string> icons = new List<string>()

{

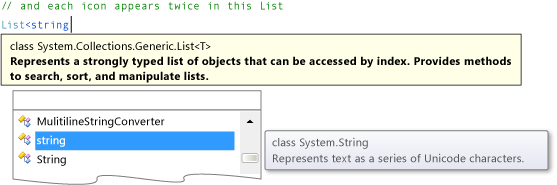
"!", "!", "N", "N", ",", ",", "k", "k",

"b", "b", "v", "v", "w", "w", "z", "z"

};

1. Go to the code editor by right-clicking **Form1.cs** in **Solution Explorer**, and then clicking **View Code** from the menu. Start typing the code shown in the previous step. If writing Visual C# code, be sure you put the code after the opening curly brace, and just after the class declaration (public partial class Form1 : Form). If writing Visual Basic code, put the code right after the class declaration (Public Class Form1).
2. When adding the **List** object, take a close look at the **IntelliSense** window that opens. The following is a Visual C# example. (Similar text appears if you add a list in Visual Basic.)

IntelliSense window



|  |
| --- |
| **NoteNote** |
| If you look at the code in small sections, it's easier to understand. Your programs can use **List** objects to keep track of many items. A list can hold numbers, true/false values, text, or other objects. You can even have a **List** object that holds other **List** objects. The items in a list are called *elements*, and each list only holds one type of element. So a list of numbers can only hold numbers—you can't add text to it. Also, you can't add numbers to a list of true/false values. |
| **NoteNote** |
| When you create a **List** object using a **new** statement, you need to tell it what you want to keep in it. That's why the tooltip at the top of the **IntelliSense** window shows the type of elements in the list. Also, that's what List<string> (in Visual C#) and List(Of String) (in Visual Basic) means: It's a **List** object that holds strings. A string is what your program uses to store text, which is what the tooltip to the right of the**IntelliSense** window is telling you. |

1. Consider why in Visual Basic a temporary array must be created first, but in Visual C#, the list can be created with one statement. This is because the Visual C# language has *collection initializers*. In Visual Basic 2010, you can use a collection initializer. However, for compatibility with the previous version of Visual Basic, we recommend using the preceding code.

|  |
| --- |
| **NoteNote** |
| When you use a collection initializer with a **new** statement, after the new **List** object is created, the program fills it with whatever is inside the curly braces. In this case, you get a list of strings named **icons**, and that list will be initialized so that it contains sixteen strings. Each of those strings is a single letter, and they all correspond to the icons that will be in the labels. So the game will have a pair of exclamation points, a pair of uppercase N letters, a pair of commas, and so on. Your **List** object will have sixteen strings in all, one for each cell in the TableLayoutPanel. |
| **NoteNote** |
| In Visual Basic, you get the same result, but first the strings are put into a temporary array, which is then converted into a **List** object. An array is similar to a list, except, for example, arrays are created with a fixed size. Lists can shrink and grow as needed, which is important in this program. |

To assign a random icon to each label

1. Before adding the following code, consider how the method works. There's a new keyword: **foreach** in Visual C# and **For Each** in Visual Basic. (One of the lines is commented out on purpose, which is explained at the end of this procedure.)

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553230(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

/// <summary>

/// Assign each icon from the list of icons to a random square

/// </summary>

private void AssignIconsToSquares()

{

// The TableLayoutPanel has 16 labels,

// and the icon list has 16 icons,

// so an icon is pulled at random from the list

// and added to each label

foreach (Control control in tableLayoutPanel1.Controls)

{

Label iconLabel = control as Label;

if (iconLabel != null)

{

int randomNumber = random.Next(icons.Count);

iconLabel.Text = icons[randomNumber];

// iconLabel.ForeColor = iconLabel.BackColor;

icons.RemoveAt(randomNumber);

}

}

}

1. Add the **AssignIconsToSquares()** method as shown in the previous step. You can put it just below the code you added in [Step 2: Add a Random Object and a List of Icons](http://msdn.microsoft.com/en-us/library/dd553237(v=vs.100).aspx).

There's something new in your **AssignIconsToSquares()** method: a **foreach** loop in Visual C# and **For Each** in Visual Basic. You use a **For Each** loop any time you want to do the same action again and again. In this case, you want to execute the same statements for every label in your TableLayoutPanel, as explained by the following code. The first line creates a variable named control that stores each control one at a time while that control has the statements in the loop executed on it.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553230(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

foreach (Control control in tableLayoutPanel1.Controls)

{

// The statements you want to execute

// for each label go here

// The statements use iconLabel to access

// each label's properties and methods

}

|  |
| --- |
| **NoteNote** |
| The names iconLabel and control are used because they are descriptive. You could replace these names with any names, and it would work exactly the same as long as you changed the name in each statement inside the loop. |

The **AssignIconsToSquares()** method goes through each **Label** control in the TableLayoutPanel and executes the same statements for each of them. Those statements pull a random icon from the list that you added in[Step 2: Add a Random Object and a List of Icons](http://msdn.microsoft.com/en-us/library/dd553237(v=vs.100).aspx). (That's why you included two of each icon in the list, so there would be a pair of icons assigned to random **Label** controls.)

Look more closely at the code that runs inside the **foreach** or **For Each** loop. This code is reproduced here.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553230(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-3)

Label iconLabel = control as Label;

if (iconLabel != null)

{

int randomNumber = random.Next(icons.Count);

iconLabel.Text = icons[randomNumber];

// iconLabel.ForeColor = iconLabel.BackColor;

icons.RemoveAt(randomNumber);

}

The first line converts the control variable to a **Label** named iconLabel. The line after that is an **if** statement that checks to make sure the conversion worked. If the conversion does work, the statements in the **if**statement run. The first line in the **if** statement creates a variable named randomNumber that contains a random number that corresponds to one of the items in the icons list. To do this, it uses the **Next** method of the **Random** object that you created earlier. The **Next** method returns the random number. This line also uses the **Count** property of the icons list to determine the range from which to choose the random number. The next line assigns one of the icon list items to the **Text** property of the label. The commented-out line is explained later in this topic. Finally, the last line in the **if** statement removes from the list the icon that has been added to the form.

Remember, if you're not sure about what some part of the code does, you can rest the mouse pointer over a code element and review the resulting tooltip.

1. You need to call the **AssignIconsToSquares()** method as soon as the program starts. If writing Visual C# code, add a statement just below the call to the **InitializeComponent()** method in the Form1 constructor, so your form calls your new method to set itself up before it's shown.

C#

public Form1()

{

InitializeComponent();

AssignIconsToSquares();

}

For Visual Basic, first add the constructor, and then add the method call to the constructor. Before the**AssignIconsToSquares()** method you just created, start by typing the code Public Sub New(). When you press the ENTER key to move to the next line, IntelliSense should make the following code appear to complete the constructor.

VB

Public Sub New()

' This call is required by Windows Form Designer

InitializeComponent()

' Add any initialization after the InitializeComponent() call

End Sub

Add the **AssignIconsToSquares()** method call so your constructor looks like the following.

VB

Public Sub New()

' This call is required by Windows Form Designer

InitializeComponent()

' Add any initialization after the InitializeComponent() call

AssignIconsToSquares()

End Sub

1. Save your program and run it. It should show a form with random icons assigned to each label.
2. Close your program, and then run it again. Now different icons are assigned to each label, as shown in the following picture.

Matching game with random icons



1. Now stop the program and uncomment that line of code inside the **For Each** loop.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553230(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-7)

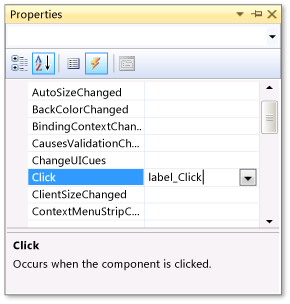
iconLabel.ForeColor = iconLabel.BackColor;

1. Click the **Save All** toolbar button to save your program, and then run it. The icons seem to have disappeared—only a blue background appears. However, the icons are randomly assigned and are still there. Because the icons are the same color as the background, they're invisible.

### To add a Click event handler to each label

1. Go to Windows Forms Designer and click the first **Label** control to select it. Then hold down the CTRL key while you click each of the other labels to select them. Be sure that every label is selected.
2. Then go to the **Events** page in the **Properties** window. Scroll down to the Click event, and type **label\_Click** in the box, as shown in the following picture.

Properties window showing Click event



1. Press ENTER. The IDE adds a Click event handler called **label\_Click()** to the code, and hooks it to each of the labels.
2. Fill in the rest of the code, as follows:

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553231(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

/// <summary>

/// Every label's Click event is handled by this event handler

/// </summary>

/// <param name="sender">The label that was clicked</param>

/// <param name="e"></param>

private void label\_Click(object sender, EventArgs e)

{

Label clickedLabel = sender as Label;

if (clickedLabel != null)

{

// If the clicked label is black, the player clicked

// an icon that's already been revealed --

// ignore the click

if (clickedLabel.ForeColor == Color.Black)

return;

clickedLabel.ForeColor = Color.Black;

}

}

|  |
| --- |
| **NoteNote** |
| You may recognize object sender at the top of the event handler from the Create a Math Quiz tutorial. You hooked up different **Label** control Click events to a single event handler method, so the same method is called no matter which label the user clicks. The method needs to know which label was clicked, so it uses the name **sender** for that **Label** control. The first line of the method tells the program that it's not just an object, but specifically a **Label** control, and that it uses the name **clickedLabel** to access its properties and methods. |

This method first checks whether **clickedLabel** was successfully converted (cast) from an object to a Label control. If unsuccessful, it has a value of null (C#) or Nothing (Visual Basic), and you don't want to execute the remainder of the code in the method. Next, the method checks the clicked label's text color using the**ForeColor** property. If it's already black, the icon's been clicked, so the method is done. (That's what the return statement does: It tells the program to stop executing the method.) If the icon hasn't been clicked, it changes its text color to black.

1. Save and run your program. You should see an empty form with a blue background. Click in the form, and one of the icons should become visible. Continue clicking in different places in the form. As you click the icons, they should appear.

### To continue or review

* To go to the next tutorial step, see [Step 5: Add Label References](http://msdn.microsoft.com/en-us/library/dd553238(v=vs.100).aspx).
* To return to the previous tutorial step, see [Step 3: Assign a Random Icon to Each Label](http://msdn.microsoft.com/en-us/library/dd553230(v=vs.100).aspx).

## Community Additions

### To add label references

1. Add label references to your form by using the following code.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553238(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

public partial class Form1 : Form

{

// firstClicked points to the first Label control

// that the player clicks, but it will be null

// if the player hasn't clicked a label yet

Label firstClicked = null;

// secondClicked points to the second Label control

// that the player clicks

Label secondClicked = null;

|  |
| --- |
| **NoteNote** |
| The reference variables look similar to the statements you used to add objects (like **Timer** objects, **List**objects, and **Random** objects) to your form. However, these statements don't cause two extra **Label**controls to appear on the form because there's no **new** in either of the two statements. Without **new**, no object is created. That's why firstClicked and secondClicked are called reference variables: They just keep track (or refer to) **Label** objects. |
| **NoteNote** |
| When a variable isn't keeping track of an object, it's set to a special value: null in Visual C# and Nothing in Visual Basic. So when the program starts, both firstClicked and secondClicked are set to null orNothing, which means that the variables aren't keeping track of anything. |

1. Modify your Click event handler to use the new firstClicked reference variable. Remove the last statement in the **label\_Click()** event handler method (clickedLabel.ForeColor = Color.Black;) and replace it with the **if** statement that follows. (Be sure you include the comment, and the whole **if** statement.)

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553238(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

/// <summary>

/// Every label's Click event is handled by this event handler

/// </summary>

/// <param name="sender">The label that was clicked</param>

/// <param name="e"></param>

private void label\_Click(object sender, EventArgs e)

{

Label clickedLabel = sender as Label;

if (clickedLabel != null)

{

// If the clicked label is black, the player clicked

// an icon that's already been revealed --

// ignore the click

if (clickedLabel.ForeColor == Color.Black)

return;

// If firstClicked is null, this is the first icon

// in the pair that the player clicked,

// so set firstClicked to the label that the player

// clicked, change its color to black, and return

if (firstClicked == null)

{

firstClicked = clickedLabel;

firstClicked.ForeColor = Color.Black;

return;

}

}

}

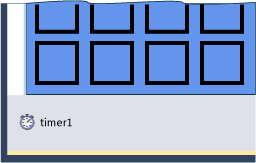
1. Save and run your program. Click one of the **Label** controls, and its icon appears.
2. Click the next **Label** control, and notice that nothing happens. The program is already keeping track of the first label that the player clicked, so firstClicked isn't equal to null in Visual C# or Nothing in Visual Basic. When your **if** statement checks firstClicked to determine if it's equal to null or Nothing, it finds that it isn't, and it doesn't execute the statements in the **if** statement. So only the first icon that's clicked turns black, and the other icons are invisible, as shown in the following picture.

Matching game showing one icon



1. Go to the Toolbox in Windows Forms Designer. Double-click **Timer** (in the **Components** category) and add a timer to the form, with its icon appearing in a gray box below the form, as shown in the following picture.

Timer



1. Click the **timer1** icon to select the timer. Set the **Interval** property to **750**, but leave the **Enabled** property set to **False**. The **Interval** property tells the timer how long to wait between ticks, so this tells the timer to wait three quarters of a second (750 milliseconds) before it fires its first Tick event. You don't want the timer to start when the program starts. Instead, you use the **Start()** method to start the timer when the player clicks the second label.
2. Double-click the **Timer** control icon in Windows Forms Designer to add the Tick event handler, as shown in the following code.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553229(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

/// <summary>

/// This timer is started when the player clicks

/// two icons that don't match,

/// so it counts three quarters of a second

/// and then turns itself off and hides both icons

/// </summary>

/// <param name="sender"></param>

/// <param name="e"></param>

private void timer1\_Tick(object sender, EventArgs e)

{

// Stop the timer

timer1.Stop();

// Hide both icons

firstClicked.ForeColor = firstClicked.BackColor;

secondClicked.ForeColor = secondClicked.BackColor;

// Reset firstClicked and secondClicked

// so the next time a label is

// clicked, the program knows it's the first click

firstClicked = null;

secondClicked = null;

}

The Tick event handler does three things: First, it stops the timer by calling the **Stop()** method. Then it uses the two reference variables, firstClicked and secondClicked, to take the two labels that the player clicked and make their icons invisible again. Finally, it resets the firstClicked and secondClicked reference variables tonull in Visual C# and Nothing in Visual Basic. That's important because that's how the program resets itself. Now it's not keeping track of any **Label** controls, and it's ready for the player's first click again.

|  |
| --- |
| **NoteNote** |
| A **Timer** object has a **Start()** method that starts the timer, and a **Stop()** method that stops it. When you set the timer's **Enabled** property to **True** in the **Properties** window, it starts ticking as soon as the program begins. But when you leave it set to **False**, it doesn't start ticking until its **Start()** method is called. |
| **NoteNote** |
| Normally, a timer fires off its Tick event again and again, using the **Interval** property to determine how many milliseconds to wait between ticks. You may have noticed how the timer's **Stop()** method is called inside the Tick event. That puts the timer into *one shot mode*, so that when the **Start()** method is called, it waits for its interval and fires off a single Tick event. |

1. To see the new timer in action, go to the code editor and add the following code to the top and bottom of the **label\_Click()** event handler method. (You're adding an **if** statement to the top, and three statements to the bottom; the rest of the method stays the same.)

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553229(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

/// <summary>

/// Every label's Click event is handled by this event handler

/// </summary>

/// <param name="sender">The label that was clicked</param>

/// <param name="e"></param>

private void label\_Click(object sender, EventArgs e)

{

// The timer is only on after two non-matching

// icons have been shown to the player,

// so ignore any clicks if the timer is running

if (timer1.Enabled == true)

return;

Label clickedLabel = sender as Label;

if (clickedLabel != null)

{

// If the clicked label is black, the player clicked

// an icon that's already been revealed --

// ignore the click

if (clickedLabel.ForeColor == Color.Black)

return;

// If firstClicked is null, this is the first icon

// in the pair that the player clicked,

// so set firstClicked to the label that the player

// clicked, change its color to black, and return

if (firstClicked == null)

{

firstClicked = clickedLabel;

firstClicked.ForeColor = Color.Black;

return;

}

// If the player gets this far, the timer isn't

// running and firstClicked isn't null,

// so this must be the second icon the player clicked

// Set its color to black

secondClicked = clickedLabel;

secondClicked.ForeColor = Color.Black;

// If the player gets this far, the player

// clicked two different icons, so start the

// timer (which will wait three quarters of

// a second, and then hide the icons)

timer1.Start();

}

}

The code at the top of the method checks whether the timer was started by checking the **Enabled** property. That way, if the player clicks the first and second **Label** controls and the timer starts, clicking a third control won't do anything.

The code at the bottom of the method sets the secondClicked reference variable so that it keeps track of the second **Label** control that the player clicked, and it sets that label's icon color to black to make it visible. Then it starts the timer in one shot mode, so that it waits for 750 milliseconds and then fires the Tick event. The timer’s Tick event handler then hides the two icons, and resets the firstClicked and secondClickedreference variables so the form is ready for the player to click another icon.

1. Save and run your program. Click an icon, and it becomes visible.
2. Click another icon. It appears briefly, and then both icons disappear. Repeat this numerous times. The form now keeps track of the first and second icons that you click, and uses the timer to pause before making the icons disappear.

To continue or review

* To go to the next tutorial step, see [Step 7: Keep Pairs Visible](http://msdn.microsoft.com/en-us/library/dd553234(v=vs.100).aspx).
* To return to the previous tutorial step, see [Step 5: Add Label References](http://msdn.microsoft.com/en-us/library/dd553238(v=vs.100).aspx).

To keep pairs visible

1. Add the following **if** statement to the **label\_Click()** event handler method, near the end just above the statement where you start the timer. Take a close look at the code while adding it to the program. Consider how the code works.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553234(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

// If the player gets this far, the timer isn't

// running and firstClicked isn't null,

// so this must be the second icon the player clicked

// Set its color to black

secondClicked = clickedLabel;

secondClicked.ForeColor = Color.Black;

// If the player clicked two matching icons, keep them

// black and reset firstClicked and secondClicked

// so the player can click another icon

if (firstClicked.Text == secondClicked.Text)

{

firstClicked = null;

secondClicked = null;

return;

}

// If the player gets this far, the player

// clicked two different icons, so start the

// timer (which will wait three quarters of

// a second, and then hide the icons)

timer1.Start();

}

}

The first line of the **if** statement you just added checks whether the icon in the first label that the player clicks is the same as the icon in the second label. If the icons are the same, the program executes the three statements between the curly braces in C# or the three statements within the **if** statement in Visual Basic. The first two statements reset the firstClicked and secondClicked reference variables so that they don't keep track of any of the labels. (You may recognize those two statements from the timer's Tick event handler.) The third statement is a return statement, which tells the program to skip the rest of the statements in the method without executing them.

If programming in Visual C#, you may have noticed that some of the code uses a single equal sign (**=**), while other statements use two equal signs (**==**). Consider why **=** is used in some places but **==** is used in other places.

This is a good example that shows the difference. Take a careful look at the code between the parentheses in the **if** statement.

VB

firstClicked.Text = secondClicked.Text

C#

firstClicked.Text == secondClicked.Text

Then look closely at the first statement in the block of code after the **if** statement.

VB

firstClicked = Nothing

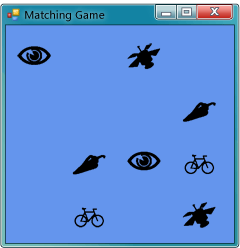
C#

firstClicked = null;

The first of those two statements checks whether two icons are the same. Because two values are being compared, the Visual C# program uses the **==** equality operator. The second statement actually changes the value (called *assignment*), setting the firstClicked reference variable equal to null to reset it. That's why it uses the **=** assignment operator instead. Visual C# uses **=** to set values, and **==** to compare them. Visual Basic uses **=** for both assignment and comparison.

1. Save and run the program, and start clicking in the form. If you click a pair that doesn't match, the timer's Tick event triggers, and both icons disappear. If you click a matching pair, the new **if** statement executes, and the return statement causes the method to skip the code that starts the timer, so the icons stay visible, as shown in the following picture.

Matching game with visible icon pairs



To add a method to verify whether the player won

1. Add a **CheckForWinner()** method to your form, as shown in the following code.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553236(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-1)

/// <summary>

/// Check every icon to see if it is matched, by

/// comparing its foreground color to its background color.

/// If all of the icons are matched, the player wins

/// </summary>

private void CheckForWinner()

{

// Go through all of the labels in the TableLayoutPanel,

// checking each one to see if its icon is matched

foreach (Control control in tableLayoutPanel1.Controls)

{

Label iconLabel = control as Label;

if (iconLabel != null)

{

if (iconLabel.ForeColor == iconLabel.BackColor)

return;

}

}

// If the loop didn’t return, it didn't find

// any unmatched icons

// That means the user won. Show a message and close the form

MessageBox.Show("You matched all the icons!", "Congratulations");

Close();

}

The method uses another **foreach** loop in Visual C# or **For Each** loop in Visual Basic to go through each label in the TableLayoutPanel. It uses the equality operator (**==** in Visual C# and **=** in Visual Basic) to check each label's icon color to verify whether it matches the background. If the colors match, the icon remains invisible, and the player hasn't matched all of the icons remaining. In that case, the program uses a **return** statement to skip the rest of the method. If the loop gets through all of the labels without executing the **return** statement, that means that all of the icons were matched. The program shows a MessageBox, and then calls the form's**Close()** method to end the game.

1. Next, have the label's Click event handler call the new **CheckForWinner()** method. Be sure that your program checks for a winner after it shows the second icon that the player clicks. Look for the line where you set the second clicked icon's color, and then call the **CheckForWinner()** method right after that, as shown in the following code.

C#

[**VB**](http://msdn.microsoft.com/en-us/library/dd553236(v=vs.100).aspx?cs-save-lang=1&cs-lang=vb#code-snippet-2)

// If the player gets this far, the timer isn't

// running and firstClicked isn't null,

// so this must be the second icon the player clicked

// Set its color to black

secondClicked = clickedLabel;

secondClicked.ForeColor = Color.Black;

// Check to see if the player won

CheckForWinner();

// If the player clicked two matching icons, keep them

// black and reset firstClicked and secondClicked

// so the player can click another icon

if (firstClicked.Text == secondClicked.Text)

{

firstClicked = null;

secondClicked = null;

return;

}

1. Save and run the program. Play the game and match all of the icons. When you win, the program displays a MessageBox (as shown in the following picture), and then closes the box.

Matching game with MessageBox



### try other features

* Replace the icons and colors with ones you choose.
* Add a grid to appear around the icons.
* Add a sound when the player finds a match, another sound when the player uncovers two icons that don't match, and a third sound when the program hides the icons again.
* Make the game more difficult by making the board bigger. (Hint: You need to do more than just add rows and columns to the TableLayoutPanel.)
* Make the game more challenging by hiding the first icon if the player is too slow and doesn't click the second icon in time.