9

Picture Boxes, Arrays

Review and Preview

In the last class, we introduced the panel control and ways to draw colored lines in a Visual C# project. We continue looking at graphics in this class.

The picture box control is studied. In particular, we use that control to display graphics files – photos, drawing, pictures. In our C# lesson, we look at a new way to declare variables and ways to count and loop. And, as a project, we build a version of the card game War.

Picture Box Control

Many times in projects, you want to display pictures or drawings saved as a graphics file on your computer. Maybe you have a little kid's program where if you type an A, an apple appears, B, a ball, and so on. Maybe you want to show a map of the United States (or some other country) for a geography lesson. Maybe you want to see some of the photos you took using a digital camera. The **picture box** is the control for that use. The picture box is selected from the toolbox. It appears as:

In Toolbox:

On Form (default properties):

Properties

The picture box control properties are:

Property	Description
Name	Name used to identify picture box control. Three letter prefix for picture box names is **pic**.
Image	Establishes the graphics file to display in the picture box.
SizeMode	Indicates how the image is displayed.
BorderStyle	Determines type of picture box border.

Left	Distance from left side of form to left side of picture box (X in properties window, expand **Position** property).
Top	Distance from top side of form to top side of picture box (Y in properties window, expand **Position** property).
Width	Width of the picture box in pixels (expand **Size** property).
Height	Height of picture box in pixels (expand **Size** property).
Enabled	Determines whether picture box can respond to user events (in run mode).
Visible	Determines whether the picture box appears on the form (in run mode).

The **Image** property is used to select the graphic file to display in the picture box and the **SizeMode** property affects how the file is displayed. Let's look at both properties.

Image Property

The picture box **Image** property specifies the graphics file to display. To set the Image property at design time, simply display the **Properties** window for the picture box control and select the Image property. An ellipsis (…) will appear. Click the ellipsis and a **Select Resource** window will appear. Select **Import** and an **Open File** dialog box will appear. Use that box to locate the graphics file to display. The picture box can display pictures stored in several different **graphics formats**. The formats we study are:

Bitmap A bitmap is an image represented by pixels (screen dots) and stored as a collection of bits in which each bit corresponds to one pixel. It usually has a **bmp** extension. You can also display icon files (**ico** extension) since they are essentially bitmap files.

GIF A GIF (Graphic Interchange Format, pronounce 'jif' like the peanut butter) file is a compressed bitmap format originally developed by the Internet provider CompuServe. Most graphics you see on the Internet are GIF files. A GIF file has a **gif** extension.

JPEG A JPEG (Joint Photographic Experts Group, pronounced 'jay-peg') file is a compressed bitmap format that is popular on the Internet and is the format usually used to store digital photographs. A JPEG file has a **jpg** extension.

These are standard graphics file types and there are other types that can be displayed. There are many programs (Paint Shop Pro by JASC, Eden Prairie, Minnesota is a good one) available that will convert a file from one type to another that you may find useful.

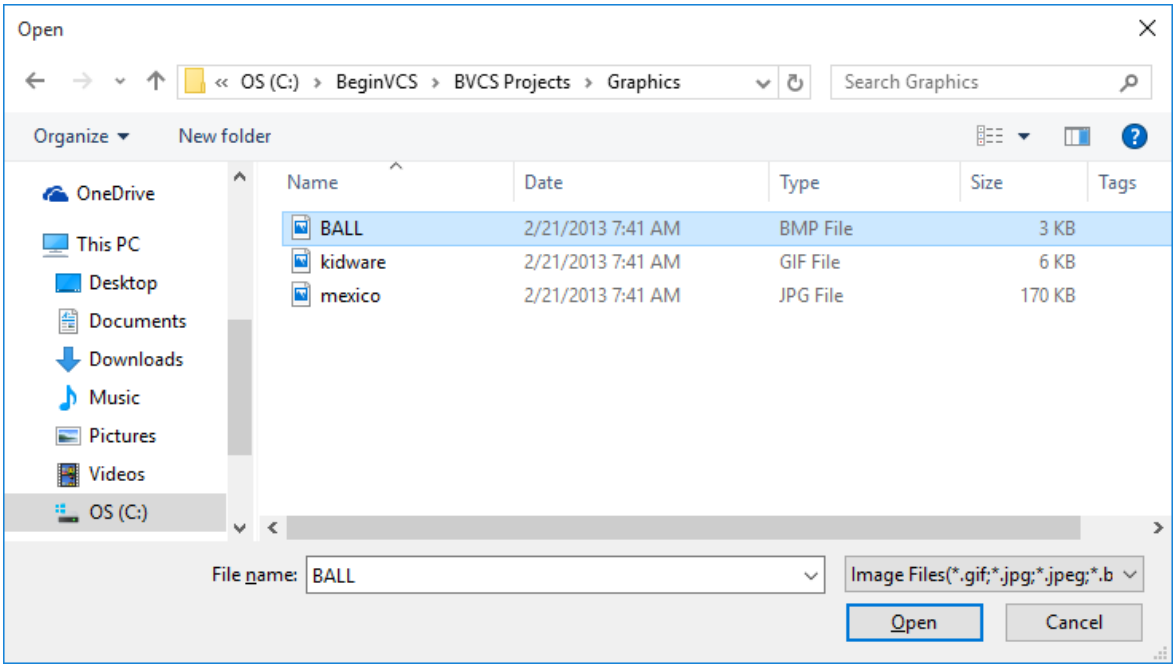
A good place to find sample bitmap and GIF files is on the Internet. To save a displayed Internet graphic as a file, right-click the graphic and choose the **Save Picture As** option (make sure it has a **bmp** or **gif** extension). If you have a digital camera, you probably have hundreds of JPEG files.

In the **\BeginVCS\BVCS Projects\Graphics** folder, we have included one file of each type for use with our examples:

ball.bmp	Bitmap picture of a ball
kidware.gif	GIF file with the logo our company (KIDware) uses on its website
mexico.jpg	Digital picture from a Mexican vacation

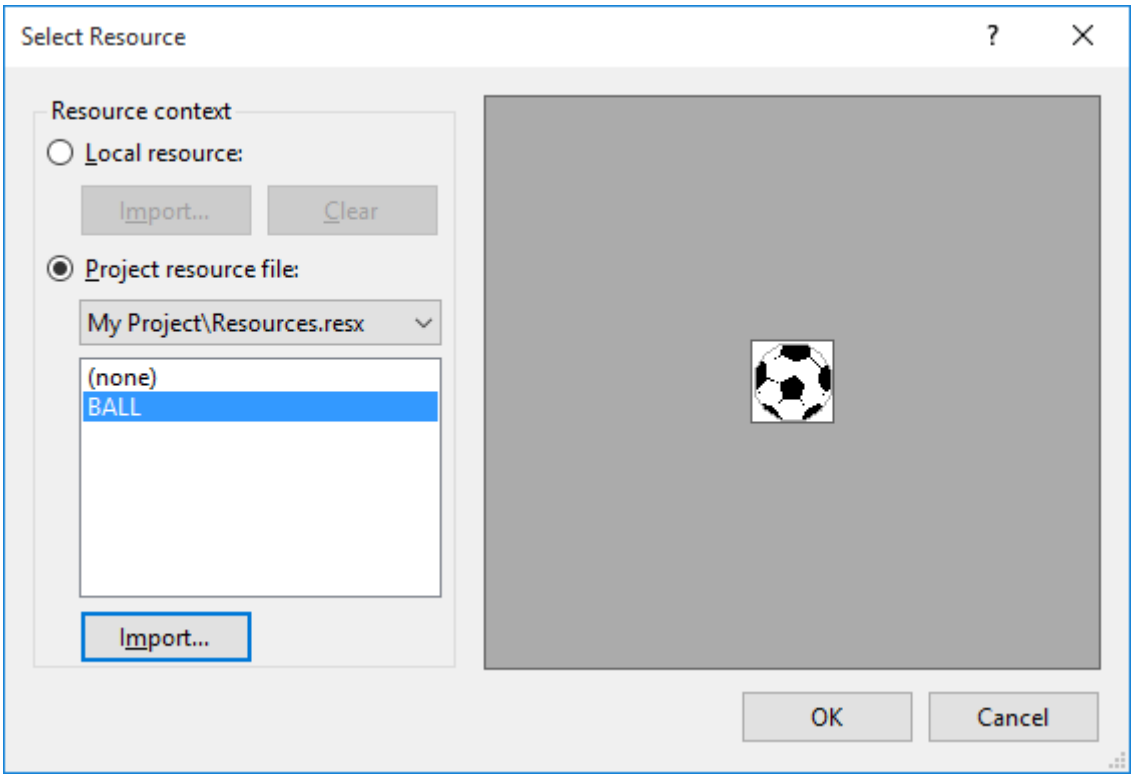
Example

Start Visual C# and start a new project. Place a picture box control on the form. Make it fairly large. Click the Image property and the ellipsis (**...**) button that appears. A **Select Resource** window will appear. Make sure the **Project resource file** radio button is selected and click the button marked **Import** and a file open window will appear. Move (as shown) to the **\BeginVCS\BVCS Projects\Graphics** folder and you will see our sample files listed:



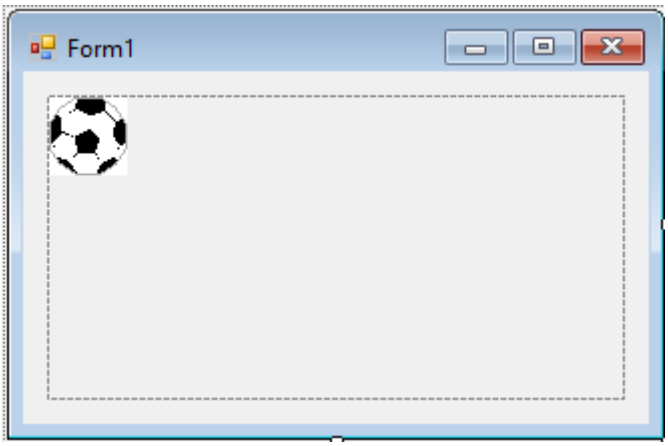
Note six file types are displayed including bitmaps, gifs, and jpegs. Other file types include metafiles (**wmf** extensions) and portable network graphics (**png** extension) – you might like to learn about these other file types on your own. One type not displayed is icon files (ico extension). To see these files, which will display just fine, you need to click **Files of type** and choose **All Files**.

Choose the **ball** bitmap file and click **Open**. You will be returned to the Select Resource window and it should look like this:



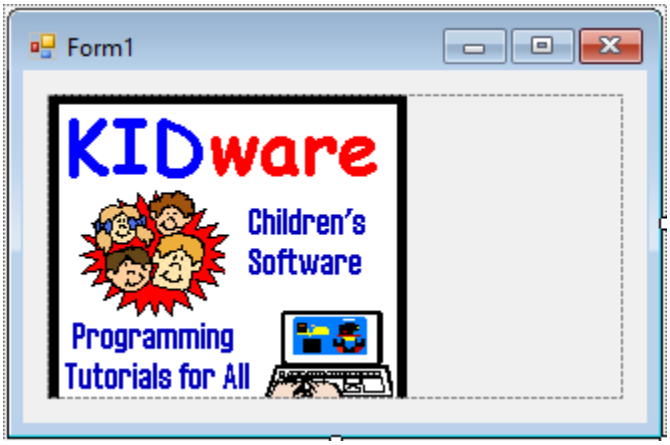
Click the **OK** button.

On your form, you should see something like this (depending on the size of your picture box):



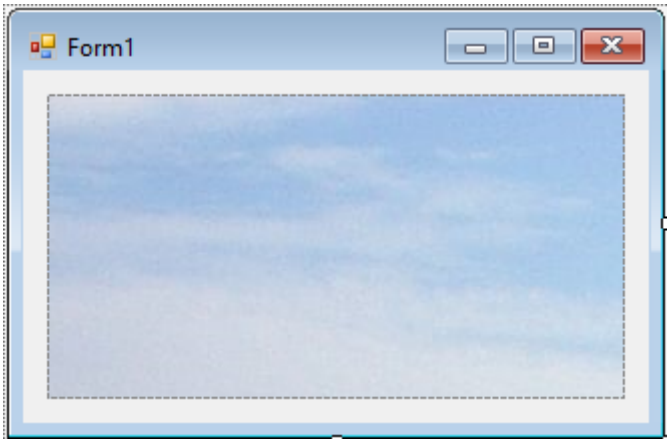
The image is in the upper left hand corner of the picture box. It appears in full-size.

Return to the properties window and choose the **kidware** logo gif file for the picture box Image property (following the same procedure using the Select Resource window):



In this example, the picture box is shorter than the graphic, so the picture is vertically “cropped.” It is located in the upper left hand corner and appears in full-size. If the picture is cropped in your example too, you can resize the picture box control to see the entire graphic.

Lastly, load and view the **mexico** JPEG file:



There’s not much to see here. The picture box is smaller than the photo, so only the sky is seen. The picture appears in full-size and is seriously cropped.

We see that the bitmap file seems to display satisfactorily. The GIF and JPEG files had cropping problems though. The **SizeMode** property of the picture box control gives us some control on how we want a graphic to display. This will help us solve some of the problems we’ve seen. We look at that property next, but first a quick look at how to remove a graphic from a picture box.

There are times you may want to delete the graphic displayed in a picture box. To do this, click Image in the properties window. In the right side of the window will be the current file (with a very tiny copy of the graphic). Select this information (double-click to highlight it) and press the keyboard **Del** key. The displayed picture will vanish and the property will read **(None)**.

SizeMode Property

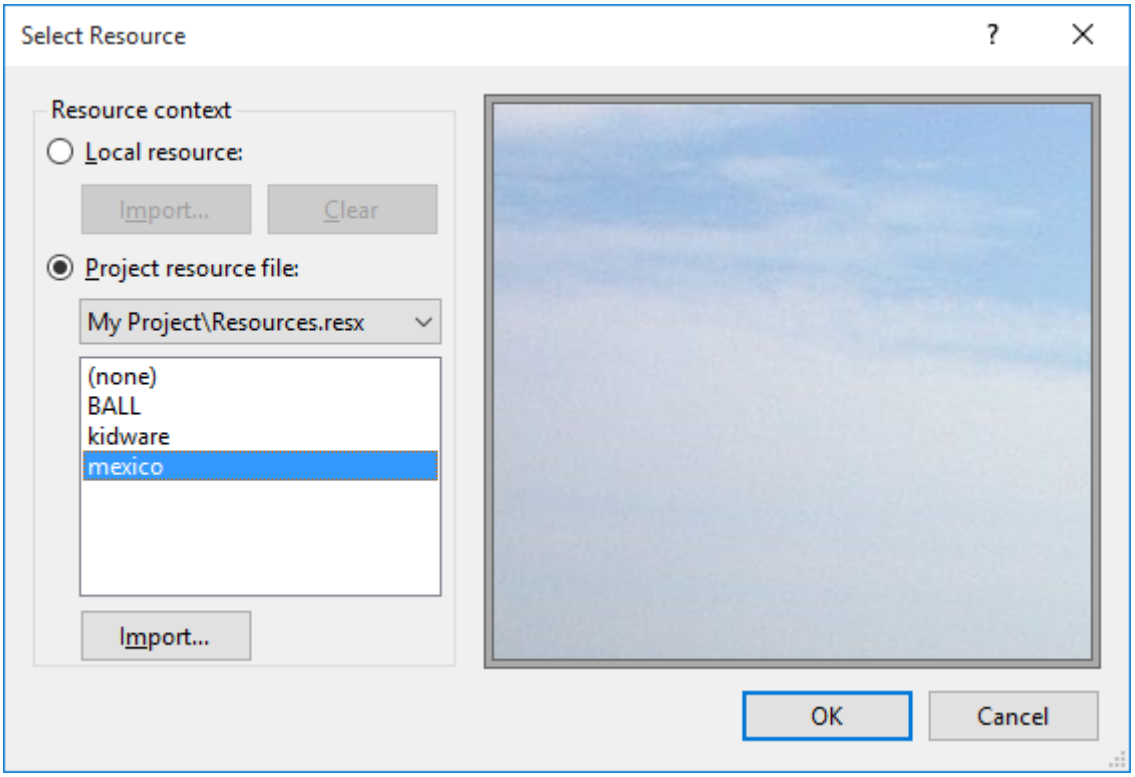
The **SizeMode** property dictates how a particular image will be displayed in a picture box. There are five possible values for this property: **Normal**, **CenterImage, StretchImage**, **AutoSize, Zoom**. The effect of each value is:

SizeMode	Effect
Normal	Image appears in original size. If picture box is larger than image, there will be blank space. If picture box is smaller than image, the image will be cropped.
CenterImage	Image appears in original size, centered in picture box. If picture box is larger than image, there will be blank space. If picture box is smaller than image, image is cropped.
StretchImage	Image will 'fill' picture box. If image is smaller than picture box, it will expand. If image is larger than picture box, it will scale down. Bitmap files do not scale nicely. JPEG and GIF files do scale nicely.
AutoSize	Reverse of StretchImage - picture box will change its dimensions to match the original size of the image. Be forewarned – some files are very large!
Zoom	Similar to StretchImage. The image will adjust to fit within the picture box, however its actual height to width ratio is maintained.

In the previous example, the **SizeMode** property had its default value of **Normal**, so all the images appeared in their original size. In the case of the bitmap file, there was lots of blank space. With the GIF and JPEG files, there was cropping. Similar results would have been seen with the **SizeMode** property changed to **CenterImage**. The most useful (in my opinion) of the **SizeMode** property choices is **StretchImage**. With this property, the image always fills the space you give it.

Example

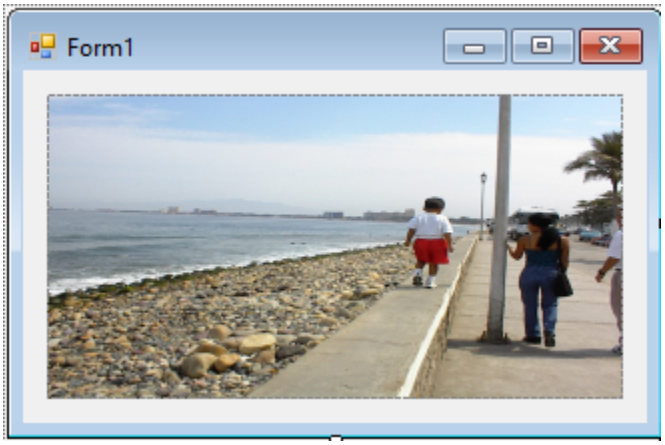
Continue the previous project. Change the **SizeMode** property of the picture box control to **CenterImage**. Reload each of the three sample graphics files. Note when you click the ellipsis next to Image in the picture box properties window, the Select Resources window will appear as:



Since each graphic already appears in the Select Resource window (these graphics have become part of the project), there is no need to reload the actual graphics files. You can choose the **Image** property directly from this window. The process is – select the desired graphic (resource) and click **OK**.

With the **CenterImage SizeMode** property, note the difference in how the files are displayed. In particular, the GIF and JPEG files are still cropped, but now they're centered in the control. For example, here is the **kidware** logo graphic with the image centered:

Change the **SizeMode** to **StretchImage**. Reload each of the sample graphics files. Notice how the graphic takes up the entire picture box. Here's the **mexico** graphic:



Try resizing the picture box control. How do the different graphics types resize? After resizing the picture box control, does the picture still look recognizable? You should find that bitmaps (in most cases) scale poorly, while GIF and JPEG graphics scale very nicely. Change the **SizeMode** to **Zoom**. Notice the graphic displays are very similar to those seen with StretchImage. The mexico graphic appears clearer, since the height to width ratio are correct:

Lastly, change the **SizeMode** to **AutoSize**. Load each graphic example and see the results. Watch out! The Mexico photo is very large. **AutoSize** should only be used when you know the size of your images and you allow for that size on your project form. If you like, try finding other graphic files on your computer and view them in the picture box with different SizeMode properties.

Events

The picture box control supports a few events. The important ones are:

Event	Description
Click	Event executed when user clicks on picture box.
MouseDown	Event executed when user presses mouse button while cursor is over picture box.
MouseMove	Event executed when user moves cursor over picture box.
MouseUp	Event executed when user releases mouse button while cursor is over picture box.

You would use the Click event when you are choosing from a group of picture box controls in a multiple choice environment. You would use the mouse events when you need to know which mouse button was pressed or released and/or where the cursor was when a mouse click, move, or release occurred.

Typical Use of Picture Box Control

The usual design steps to use a picture box control for displaying a graphic file are:

- Set the **Name** and **SizeMode** property (most often, **StretchImage**).
- Set **Image** property, either in design mode or at run-time.

C# - The Sixth Lesson

In this C# lesson, we look at ways to store large numbers of variables, a technique for counting, and some code for shuffling a deck of cards (or randomly sorting a list of numbers).

Variable Arrays

Your local school principal has recognized your great C# programming skills and has come for your help. Everyone (352 students) in the school has just taken a C# skills test. The principal wants you to write a program that stores each student's name and score. The program should rank (put in order) the scores and compute the average score. The code to do this is not that hard. The problem we want to discuss here is how do we declare all the variables we need? To write this test score program, you need 352 **string** variables to store student names and 352 **int** variables to store student scores. We are required to declare every variable we use. Do you want to type 704 lines of code something like this?:

```
string student1;
string student2;
string student3;
  .
  .
string student352;
int score1;
int score2;
int score3;
  .
  .
int score352;
```

I don't think so.

C# provides a way to store a large number of variables under the same name - **variable arrays**. Each variable in an array, called an **element**, must have the same data type, and they are distinguished from each other by an array **index**. A variable array is declared in a way similar to other variables. To indicate the variable is an array, you use two square brackets (**[ ]**) after the type. Square brackets are used a lot with arrays. At the same time you declare an array, it is good practice to create it using the **new** keyword. For 352 student names and 352 student scores, we declare and create the needed arrays using:

```
string[] student = new string[352];
int[] score = new int[352];
```

The number in brackets is called the array **dimension**. These two lines have the same effect as the 704 declaration lines we might have had to write! And, notice how easy it would be to add 200 more variables if we needed them. You can also declare and create an array in two separate statements if you prefer. For the student name array, that code would be:

```
string[] student;  // the declaration;
student = new string[352];  // the creation
```

We now have 352 **student** variables (**string** type) and 352 **score** variables (**int** type) available for our use. A very important concept to be aware of is that C# uses what are called **zero-based** arrays. This means array indices begin with 0 and end at the dimension value minus 1, in this case 351. Each variable in an array is referred to by its declared name and index. The first student name in the array would be **student[0]** and the last name would be **student[351]**, not student[352]. If you try to refer to student[352], you will get a run-time error saying an array value is out of bounds. This is a common mistake! When working with arrays in C#, always be aware they are zero-based.

As an example of using an array, to assign information to the student with index of 150 (actually, the 151st student in the array because of the zero base), we could write two lines of code like this:

```
student[150] = “Billy Gates”;
score[150] = 100;
```

Array variables can be used anywhere regular variables are used. They can be used on the left side of assignment statements or in expressions. To add up the first three test scores, you would write:

```
sum = score[0] + score[1] + score[2];
```

Again, notice the first score in the array is **score[0]**, not **score[1]**. I know this is confusing, but it’s something you need to remember. We still need to provide values for each element in each array, but there are also some shortcuts we can take to avoid lots of assignment statements. One such shortcut, the **for** loop, is examined next. You will find variable arrays are very useful when working with large numbers (and sometimes, not so large numbers) of similar variables.

C# for Loops

A common computer programming task is counting. We might like to execute some C# code segment a particular number of times - we would need to count how many times we executed the code. In the school score example from above, we need to go through all 352 scores to compute an average. C# offers a convenient way to do counting: the **for** loop.

The C# **for** loop has this unique structure:

```
for (initialization; expression; update)
{
      [C# code block to execute]
}
```

After the word **for** are three parts separated by semicolons: **initialization**, **expression**, and **update**. The first, **initialization**, is a step executed once and is used to initialize a counter variable (usually an **int** type). A very common initialization would start a counter **i** at zero:

```
i = 0
```

The second part, **expression**, is a step executed before each iteration (repetition) of the code in the loop. If expression is true, the code is executed; if false, program execution continues at the line following the end of the for loop. A common expression would be:

```
i < iMax
```

The final part, **update**, is a step executed after each iteration of the code in the loop; it is used to update the value of the counter variable. A common update would be:

```
i = i + 1
```

Using these example steps, a for loop appears as:

```
for (i = 0; i < iMax; i = i +1)
{
      [C# code block to execute]
}
```

In this example, the counter **i** is initialized at **0** and is then incremented (changed) by **1** each time the program executes the loop. For each execution of the loop, any code between the two curly braces is repeated. The loop is repeated as long as **i** remains smaller than **iMax**. When the loop is completed, program execution continues after the closing brace. To leave the loop before completion, you can use the **break** statement introduced with the switch structure.

A few examples should clear things up. Assume we want to set the value of 10 elements of some array, **myArray[10]**, to 0. The for loop that would accomplish this task is:

```
For (i = 0; i < 10; i = i + 1)
{
    myArray[i] = 0;
}
```

In this loop, the counter variable **i** (declared to be an **int** variable prior to this statement) is initialized at 0. With each iteration, i is incremented by one. The loop is repeated as long as i remains smaller than 10 (remember myArray[9] is the last element of the array).

How about a rocket launch countdown? This loop will do the job:

```
For (i = 10; i <= 0; i = i - 1)
{
      [C# code block for the countdown]
}
```

Here i starts at 10 and goes down by 1 (i = i -1) each time the loop is repeated. Yes, you can decrease the counter. And, you can have counter increments that are not 1. This loop counts from 0 to 200 by 5's:

```
For (i = 0; i <= 200; i = i + 5)
{
      [C# code block to execute]
}
```

In each of these examples, it is assumed that i has been declared prior to these loops.

How about averaging the scores from our student example. This code will do the job:

```
scoreSum = 0;
for (studentNumber = 0; studentNumber < 352; studentNumber =
studentNumber + 1)
{
    scoreSum = scoreSum + score[StudentNumber];
}
average = scoreSum / 300;
```

(Again, it is assumed that all variables have been declared to have the proper type). To find an average of a group of numbers, you add up all the numbers then divide by the number of numbers you have. In this code, scoreSum represents the sum of all the numbers. We set this to zero to start. Then, each time through the loop, we add the next score to that “running“ sum. The loop adds up all 352 scores making use of the score array. The first time through it adds in score[0], then score[1], then score[2], and so on, until it finishes by adding in score[351]. Once done, the average is computed by dividing scoreSum by 352. Do you see how the for loop greatly simplifies the task of adding up 352 numbers? This is one of the shortcut methods we can use when working with arrays. Study each of these examples so you have an idea of how the for loop works. Use them when you need to count.

Before leaving the for loop, let’s look at one more thing. A very common update to a counter variable is to add one (increment) or subtract one (decrement). C# has special increment and decrement operators that do just that. To add one to a variable named **counterVariable**, you can simply write:

```
counterVariable++;
```

This statement is equivalent to:

```
counterVariable = counterVariable + 1;
```

Similarly, the decrement operator:

```
counterVariable--;
```

Is equivalent to:

```
counterVariable = counterVariable – 1;
```

The increment and decrement operators are not limited to for loops. They can be used anywhere they are needed in a C# program.

Block Level Variables

Let's address another issue. Notice, at a minimum, the for loop requires the declaration of one variable, the loop counter, usually an **int** type variable. This variable is only used in the code block associated with this loop - it's value is usually of no use anywhere else. When we declare a variable in the general declarations area of the code window, as we have been doing, its value is available to all event methods. We say such variables have **form level scope**. Such declarations are not necessary with for loop counters and it becomes a headache if you have lots of for loops. Loop counters can be declared in the initialization part of the for statement. We give these variables **block level scope** - their value is only known within that loop's code block.

As an example of declaring block level variables, look at a modification to the student average example:

```
scoreSum = 0;
for (int studentNumber = 0; studentNumber < 352;
studentNumber++)
{
  scoreSum = scoreSum + score[StudentNumber];
}
average = scoreSum / 300;
```

Notice how the counter (studentNumber) is declared and initialized, all in one step, in the **for** statement. This is perfectly acceptable in C# - whenever, you declare a variable, you can also assign an initial value. Once the for loop is complete, the value of studentNumber is no longer known or available. As you write C# code, you will often give your loop variables such block level scope. Also, notice how we've modified this example to include the increment operator (++).

Method Level Variables

In addition to **block** level and **form** level variables, there is one other level of variable scope we can use – **method** level variables. If a variable only has used within a particular method, there is no need to declare it in the general declarations area. Variables with method level scope are declared immediately following the opening curly brace for a method.

As an example of declaring method level variables, assume we have a button control (named **btnAverage**) on a form that computes the student average score in our example. The **btnAverage_Click** method procedure would look like this:

```
private void btnAverage_Click(object sender, EventArgs e)
{
    int scoreSum;
    scoreSum = 0;
    for (int studentNumber = 0; studentNumber < 352;
studentNumber++)
    {
        scoreSum = scoreSum + score[StudentNumber];
    }
    average = scoreSum / 300;
}
```

In this example, **scoreSum** is only used and needed in this method, hence is declared as a method level variable. The variable **average** should be declared in the general declarations area so it has form level scope and is available everywhere in your project. As you write C# code, decide whether you want your variables to have **block** level, **method** level or **form** level scope and declare them in the proper area in the code window.

Shuffle Routine

Let's use our new knowledge of arrays and for loops to write a very useful method. A common task in any computer program is to randomly sort a list of consecutive integer values. Why would you want to do this? Say you have four answers in a multiple choice quiz. Randomly sort the integers 1, 2, 3, and 4, so the answers are presented in random order. Or, you have a quiz with 30 questions. Randomly sort the questions for printing out as a worksheet. Or, the classic application is shuffling a deck of standard playing cards (there are 52 cards in such a deck). In that case, you can randomly sort the integers from 0 to 51 to "simulate" the shuffling process. Let's build a "shuffle" routine. We call it a shuffle routine, recognizing it can do more than shuffle a card deck. Our routine will sort any number of consecutive integers.

Usually when we need a computer version of something we can do without a computer, it is fairly easy to write down the steps taken and duplicate them in C# code. We've done that with the projects built so far in this course. Other times, the computer version of a process is easy to do on a computer, but hard or tedious to do off the computer. When we shuffle a deck of cards, we separate the deck in two parts, then interleaf the cards as we fan each part. I don't know how you could write C# code to do this. There is a way, however, to write C# code to do a shuffle in a more tedious way (tedious to a human, easy for a computer).

We will perform what could be called a "one card shuffle." In a one card shuffle, you pull a single card (at random) out of the deck and lay it aside on a pile. Repeat this 52 times and the cards are shuffled. Try it! I think you see this idea is simple, but doing a one card shuffle with a real deck of cards would be awfully time-consuming. We'll use the idea of a one card shuffle here, with a slight twist. Rather than lay the selected card on a pile, we will swap it with the bottom card in the stack of cards remaining to be shuffled. This takes the selected card out of the

deck and replaces it with the remaining bottom card. The result is the same as if we lay it aside.

Here's how the shuffle works with n numbers:

- Start with a list of n consecutive integers.
- Randomly pick one item from the list. Swap that item with the last item. You now have one fewer items in the list to be sorted (called the remaining list), or n is now n - 1.
- Randomly pick one item from the remaining list. Swap it with the item on the bottom of the remaining list. Again, your remaining list now has one fewer items.
- Repeatedly remove one item from the remaining list and swap it with the item on the bottom of the remaining list until you have run out of items. When done, the list will have been replaced with the original list in random order.

Confusing? Let's show a simple example with n = 5 (a very small deck of cards).

The starting list is (with 5 remaining items):

1 2 3 4 5

Remaining List

We want to pick one item, at random, from this list. Using the C# random number generator, we would choose a random number from 1 to 5. Say it was 3. We take the third item in the list (the 3) and swap it with the last item in the list (the 5). We now have:

1 2 5 4 3

Remaining List

There are 4 items in the remaining list. Pick a random number from 1 to 4 - say it's 4. The fourth item in the remaining list is 4. Swap it with the last item in the remaining list. Wait a minute! The last item in the remaining list is the 4. In this case, we swap it with itself, or it stays put. If the random number was something other than 4, there really would have been a swap here. We now have:

1 2 5 4 3

Remaining List

There are 3 items in the remaining list. Pick a random number from 1 to 3 - say it's 1. The first item in the list is 1. Swap the 1 with the last item in the remaining list (the 5), giving us:

5 2 1 4 3

Remaining List

There are 2 items in the remaining list. Pick a random number from 1 to 2 - say it's 1. The first item in the list is 5. Swap the 5 with the last item in the remaining list (the 2), giving us the final result, the numbers 1 to 5 randomly sorted:

2 5 1 4 3

Pretty neat how this works, huh?

We want to describe the one card shuffle with C# code. Most of the code is straightforward. The only question is how to do the swap involved in each step. This swap is easy on paper. How do we do a swap in C#? Actually, this is a common C# task and is relatively simple. At first thought, to swap variable aVariable with variable bVariable, you might write:

```
aVariable = bVariable;
bVariable = aVariable;
```

The problem with this code is that when you replace aVariable with bVariable in the first statement, you have destroyed the original value of aVariable. The second statement just puts the newly assigned aVariable value (bVariable) back in bVariable. Both aVariable and bVariable now have the original bVariable value! Actually, swapping two variables is a three step process. First, put aVariable in a temporary storage variable (make it the same type as aVariable and bVariable). Then, replace aVariable by bVariable. Then, replace bVariable by the temporary variable (which holds the original aVariable value). If tVariable is the temporary variable, a swap of aVariable and bVariable is done using:

```
tVariable = aVariable;
aVariable = bVariable;
bVariable = tVariable;
```

You use swaps like this in all kinds of C# applications.

Now, we'll see the C# code that uses a one card shuffle to randomly sort N consecutive integer values. When done the random list of integers is in the array **numberList**, which should be declared in the general declarations area of your project with the proper dimension. Also declare the variable, **numberOfItems**, which is the length of the list. For a deck of cards, these declarations would be:

```
int[] numberList = new int[52];
int numberOfItems;
```

You need to make sure to assign a value (52) to **numberOfItems** somewhere, most likely in the form **Load** procedure. We need a random number object (**myRandom**) with method level scope to do all the random number generation. And four variables will have block level scope within the particular for loops implementing the shuffle:

loopCounter - integer loop counter variable
remaining - integer loop variable giving number of items in remaining list
itemPicked - integer variable giving item picked in remaining list
tempValue - temporary integer variable used for swapping

One note – recall arrays in C# are zero-based. In this code, if you ask it to shuffle n consecutive integers, the indices on the returned array range from 0 to n – 1 and the randomized integers will also range from 0 to n – 1, not 1 to n. If you need integers from 1 to n, just simply add 1 to each value in the returned array! The code is:

```
// Variable declarations (put at top of method)
Random myRandom = new Random();

// One card shuffle code
// initialize NumberList
for (int loopCounter = 0; loopCounter < numberOfItems;
loopCounter++)
{
    numberList[loopCounter] = loopCounter;
}
// Work through remaining values
// Start at numberOfItems and swap one value
// at each for loop step
// After each step, remaining is decreased by 1
for (int remaining = numberOfItems; remaining >= 1;
remaining--)
{
    // Pick item at random
    int itemPicked = myRandom.Next(remaining);
    // Swap picked item with bottom item
    int tempValue = numberList[itemPicked];
    numberList[itemPicked] = numberList[remaining - 1];
    numberList[remaining - 1] = tempValue;
}
```

Study this code and see how it implements the procedure followed in the simple five number example. It's not that hard to see. Understanding how such code works is a first step to becoming a good C# programmer. Notice this bit of code uses everything we talked about in this class' C# lesson: arrays, for loops, and block and method level variables.

Project - Card Wars

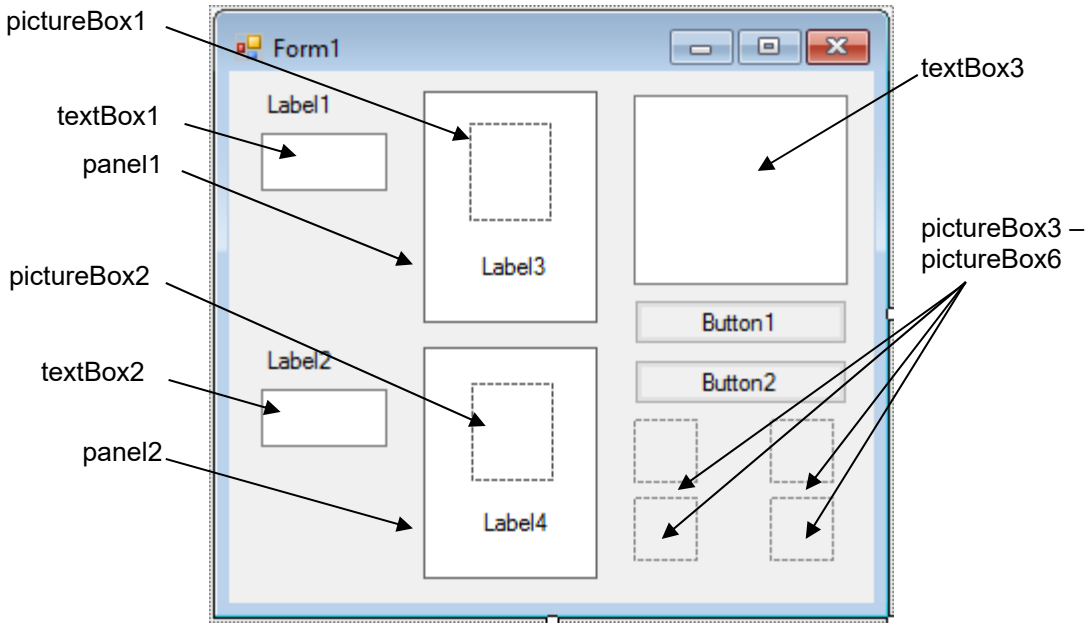
In this project, we create a simplified version of the kid's card game - War. You play against the computer. You each get half a deck of cards (26 cards). Each player turns over one card at a time. The one with the higher card wins the other player's card. The one with the most cards at the end wins. Obviously, the shuffle routine will come in handy here. We call this project Card Wars! This project is saved as **CardWars** in the projects folder **(\BeginVCS\BVCS Projects**).

Project Design

We will use a panel control to represent the outline of each player's card. A label control will show the card's value and a picture box control will display the card's suit (hearts, diamonds, clubs, spades). A button will control starting a new game or drawing a new card, depending on game state. Another button will control stopping the game, if playing, or stopping the program, if not playing. The current score (number of cards each player has) will be displayed in a labeled text box control.

Place Controls on Form

Start a new project in Visual C#. Place two panel controls on the form. Size them to represent the two displayed cards. Place a label and picture box control in each panel. Add two buttons to the form. Add two labels and two text boxes that will be used for the scoring system. Add a large text box to tell us when the game is over. And, add four more picture boxes that will be used to hold the images for each card suit. When done, my form looks like this:



Set Control Properties

Set the control properties using the properties window:

Form1 Form:

Property Name	Property Value
Text	Card Wars
FormBorderStyle	Fixed Single
StartPosition	CenterScreen

panel1 Panel:

Property Name	Property Value
Name	pnlPlayer
BackColor	White
BorderStyle	FixedSingle

panel2 Panel:

Property Name	Property Value
Name	pnlComputer
BackColor	White
BorderStyle	FixedSingle

picture Box1 Picture Box:

Property Name	Property Value
Name	picPlayer
SizeMode	StretchImage

picture Box2 Picture Box:

Property Name	Property Value
Name	picComputer
SizeMode	StretchImage

pictureBox3 Picture Box:

Property Name	Property Value
Name	picHeart
Image	Heart.ico (in **\BeginVCS\BVCS Projects\CardWars** folder)
SizeMode	AutoSize
Visible	False

pictureBox4 Picture Box:

Property Name	Property Value
Name	picDiamond
Image	Diamond.ico (in **\BeginVCS\BVCS Projects\CardWars** folder)
SizeMode	AutoSize
Visible	False

pictureBox5 Picture Box:

Property Name	Property Value
Name	picClub
Image	Club.ico (in **\BeginVCS\BVCS Projects\CardWars** folder)
SizeMode	AutoSize
Visible	False

pictureBox6 Picture Box:

Property Name	Property Value
Name	picSpade
Image	Spade.ico (in **\BeginVCS\BVCS Projects\CardWars** folder)
SizeMode	AutoSize
Visible	False

The Visible properties for these four picture box controls (picHeart, picDiamond, picClub, picSpade) are purposely False. We don't want them to show up on the form in run mode - we just want to use their stored picture for our card displays. This is done a lot in Visual C#. When setting the Image property, in the Open File Dialog, you will need to make sure you view **All Files** and not just the **Image Files**. Icon files will not appear unless you make this change.

label1 Label:

Property Name	Property Value
Name	lblYou
Text	You
Font Size	10

textBox1 Text Box:

Property Name	Property Value
Name	txtYouScore
Text	0
Font Size	12
Font Style	Bold
ReadOnly	True
TextAlign	Center
BackColor	White

label2 Label:

Property Name	Property Value
Name	lblComp
Text	Computer
Font Size	10

textBox2 Text Box:

Property Name	Property Value
Name	txtCompScore
Text	0
Font Size	12
Font Style	Bold
ReadOnly	True
TextAlign	Center
BackColor	White

label3 Label:

Property Name	Property Value
Name	lblPlayer
Text	[Blank]
Font Size	18
Font Style	Bold
TextAlign	MiddleCenter

label4 Label:

Property Name	Property Value
Name	lblComputer
Text	[Blank]
Font Size	18
Font Style	Bold
TextAlign	MiddleCenter

textBox3 Text Box:

Property Name	Property Value
Name	txtOver
Text	Game Over
Font Size	14
Font Style	Bold
ReadOnly	True
TextAlign	Center
BackColor	White
ForeColor	Red

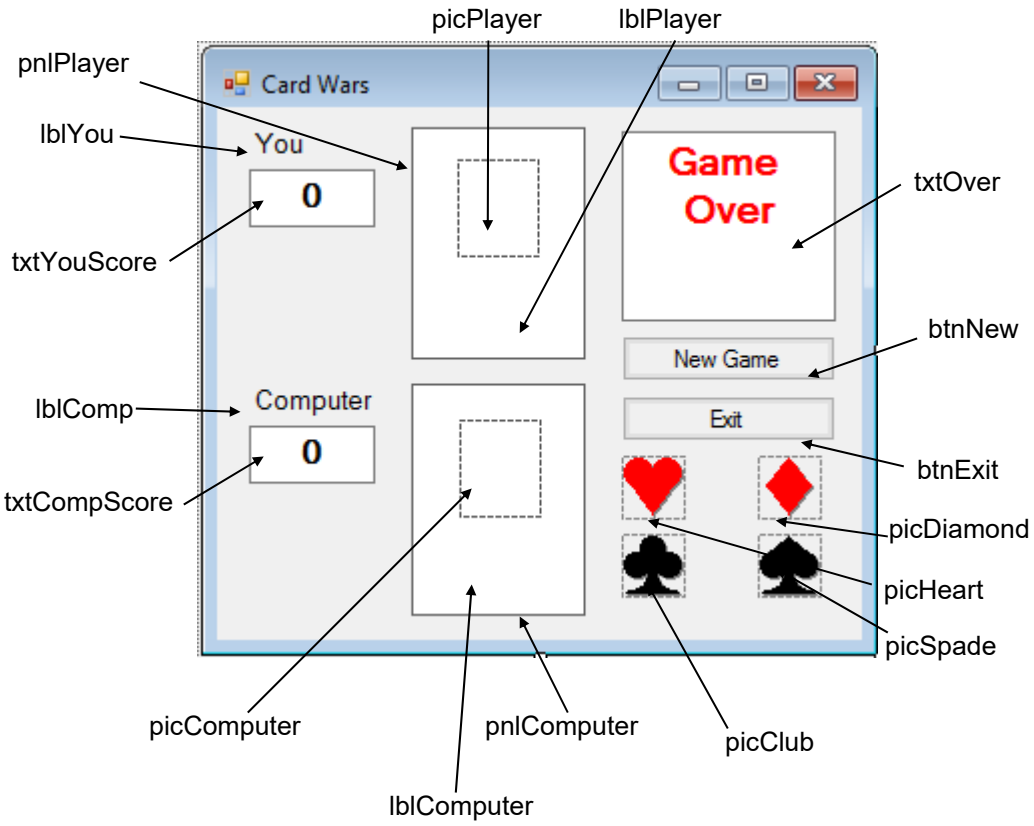
button1 Button:

Property Name	Property Value
Name	btnNew
Text	New Game

button2 Button:

Property Name	Property Value
Name	btnExit
Text	Exit

When done, my form looks like this:



Write Event methods

The idea of this game is quite simple. You click the **New Game** button to start. This shuffles the cards, resets the scores to zero and changes the button's Text to **Next Card**. It also changes the **Exit** button Text to **Stop** (for stopping the current game). A card for you (upper card) and a card for the computer (lower card) are displayed. The computer decides which card is higher. The player with the higher card gets two points. If it's a tie, each player gets one point. Scores are displayed under **You** (txtYouScore has your score) and **Computer** (txtCompScore has computer's score). Click **Next Card**. A new card is displayed for each player and the scores updated. Continue clicking **Next Card** until the game is over (each player has shown 26 cards). At that point, the 'Game Over' message is displayed and the button captions are reset to their original values. By checking the score, a winner can be determined. You can stop the game early by clicking **Stop**. There are only two event methods - one for **btnNew_Click** and one for **btnExit_Click**. Before looking at these events, let's look at needed variables.

We only need two variables (well, really 53, but, arrays help out) for this project. The first is an integer array (**cardNumber**) that has the 52 shuffled numbers representing each card in the deck. The first half (cardNumber[0] – cardNumber[25]) will be your cards while the second half (cardNumber[26] – cardNumber[51]) will be the computer's. The second variable is **cardIndex**, an integer indicating which card to display. Open the code window and declare these variables in the **general declarations** area:

```
int [] cardNumber = new int[52];
int cardIndex;
```

Now, let's outline the steps involved in the **btnNew_Click** event. First, we are letting this command button have two purposes. It either starts a new game (**Text** is **New Game**) and or gets a new card (**Text** is **Next Card**). So, the Click event has two segments. If Text is New Game, the steps are:

- Hide 'Game Over' notice
- Set btnNew Text to "Next Card"
- Set btnExit Text to "Stop"
- Set scores to zero
- Shuffle cards
- Initialize cardIndex to zero
- Display first card for each player
- Compare cards - update score

If Text is Next Card, the steps are:

- Display two new cards
- Compare displayed cards - update scores
- Increment CardIndex
- If there are no cards left, stop game - display 'Game Over' message, change button captions. Otherwise, wait for click on Next Card button.

Most of these steps are easily done now that we know how to shuffle a deck of cards. The only tough part is deciding how to display and compare cards. Let's look at that in some detail.

Displaying a card consists of answering two questions: what is the card suit and what is the card value? The four suits are hearts, diamonds, clubs, and spades. The thirteen card values, from lowest to highest, are: 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack (J), Queen (Q), King (K), Ace (A). We've seen in our shuffle routine that a card number will range from 0 to 51. How do we translate that card number to a card suit and value? (Notice the distinction between card **number** and card **value -** card number ranges from 0 to 51, card value can only range from 2 to Ace.) We need to develop some type of translation rule. This is done all the time in C#. If the number you compute with or work with does not directly translate to information you need, you need to make up rules to do the translation. For example, the numbers 1 to 12 are used to represent the months of the year. But, these numbers tell us nothing about the names of the month. We need a rule to translate each number to a month name.

We know we need 13 of each card suit. Hence, an easy rule to decide suit is: cards numbered 0 - 12 are hearts, cards numbered 13 - 25 are diamonds, cards numbered 26 - 38 are clubs, and cards numbered 39 - 51 are spades. Suit is represented on the displayed card by the two picture boxes: picPlayer (your card) and picComputer (computer's card). For card values, lower numbers should represent lower cards. A rule that does this for each number in each card suit is:

Card Numbers

Hearts	Diamonds	Clubs	Spades	Card Value
0	13	26	39	Two
1	14	27	40	Three
2	15	28	41	Four
3	16	29	42	Five
4	17	30	43	Six
5	18	31	44	Seven
6	19	32	45	Eight
7	20	33	46	Nine
8	21	34	47	Ten
9	22	35	48	Jack
10	23	36	49	Queen
11	24	37	50	King
12	25	38	51	Ace

As examples, notice card 22 is a Jack of Diamonds. Card 30 is a 6 of Clubs. The card values are displayed in the lblPlayer and lblComputer label controls. We now can display cards. How do we compare them?

Card comparisons must be based on a numerical value, not displayed card value - it's difficult to check if K is greater than 7, though it can be done. So, one last rule is needed to relate card value to numerical value. It's a simple one - start with a 2 having a numerical value of 0 (lowest) and go up, with an Ace (A) having a numerical value of 12 (highest). This makes numerical card comparisons easy. Notice hearts card numbers already go from 0 to 12. If we subtract 13 from diamonds numbers, 26 from clubs numbers, and 39 from spades numbers, each of those card numbers will also range from 0 to 12. This gives a common basis for comparing cards. This all may seem complicated, but look at the C# code and you'll see it really isn't.

The C# code that implements the **btnNew_Click** event method is:

```
private void btnNew_Click(object sender, EventArgs e)
{
    // Method level variables
    Random myRandom = new Random();
    int yourNumber = 0; // Your card number
    int computerNumber = 0; // Computer card number
    if (btnNew.Text == "New Game")
    {
        // New game clicked
        txtOver.Visible = false;
        btnNew.Text = "Next Card";
        btnExit.Text = "Stop";
        // Zero out scores
        txtYouScore.Text = "0";
        txtCompScore.Text = "0";
        // Shuffle cards using one card shuffle code
        // Initialize CardNumbers
        for (int loopCounter = 0; loopCounter < 52;
loopCounter++)
        {
            cardNumber[loopCounter] = loopCounter;
        }
        // Work through remaining values
        // Start at 52 and swap one value
        // at each for loop step
        // After each step, remaining is decreased by 1
        for (int remaining = 52; remaining >= 1; remaining--)
        {
            // Pick item at random
            int itemPicked = myRandom.Next(remaining);
            // Swap picked item with bottom item
            int tempValue = cardNumber[itemPicked];
            cardNumber[itemPicked] = cardNumber[remaining -
1];
            cardNumber[remaining - 1] = tempValue;
        }
        // Set CardIndex to zero
        cardIndex = 0;
    }
    // Display cards
    // Display your card's suit
    // Determine your card's number for comparisons
    if (cardNumber[cardIndex] >= 0 && cardNumber[cardIndex]
<= 12)
```

```
    {
        picPlayer.Image = picHeart.Image;
        yourNumber = cardNumber[cardIndex];
    }
    else if (cardNumber[cardIndex] >= 13 &&
cardNumber[cardIndex] <= 25)
    {
        picPlayer.Image = picDiamond.Image;
        yourNumber = cardNumber[cardIndex] - 13;
    }
    else if (cardNumber[cardIndex] >= 26 &&
cardNumber[cardIndex] <= 38)
    {
        picPlayer.Image = picClub.Image;
        yourNumber = cardNumber[cardIndex] - 26;
    }
    else if (cardNumber[cardIndex] >= 39 &&
cardNumber[cardIndex] <= 51)
    {
        picPlayer.Image = picSpade.Image;
        yourNumber = cardNumber[cardIndex] - 39;
    }
    // Display your card's value
    switch (yourNumber)
    {
        case 9:
            lblPlayer.Text = "J";
            break;
        case 10:
            lblPlayer.Text = "Q";
            break;
        case 11:
            lblPlayer.Text = "K";
            break;
        case 12:
            lblPlayer.Text = "A";
            break;
        default:
            lblPlayer.Text = Convert.ToString(yourNumber + 2)
+ " ";
            break;
    }
    // Display computer's card suit
    // Determine computer's number for comparisons
    if (cardNumber[cardIndex + 26] >= 0 &&
cardNumber[cardIndex + 26] <= 12)
    {
```

```
        picComputer.Image = picHeart.Image;
        computerNumber = cardNumber[cardIndex + 26];
    }
    else if (cardNumber[cardIndex + 26] >= 13 &&
cardNumber[cardIndex + 26] <= 25)
    {
        picComputer.Image = picDiamond.Image;
        computerNumber = cardNumber[cardIndex + 26] - 13;
    }
    else if (cardNumber[cardIndex + 26] >= 26 &&
cardNumber[cardIndex + 26] <= 38)
    {
        picComputer.Image = picClub.Image;
        computerNumber = cardNumber[cardIndex + 26] - 26;
    }
    else if (cardNumber[cardIndex + 26] >= 39 &&
cardNumber[cardIndex + 26] <= 51)
    {
        picComputer.Image = picSpade.Image;
        computerNumber = cardNumber[cardIndex + 26] - 39;
    }
    // Display computer card's value
    switch (computerNumber)
    {
        case 9:
            lblComputer.Text = "J";
            break;
        case 10:
            lblComputer.Text = "Q";
            break;
        case 11:
            lblComputer.Text = "K";
            break;
        case 12:
            lblComputer.Text = "A";
            break;
        default:
            lblComputer.Text =
Convert.ToString(computerNumber + 2) + " ";
            break;
    }
    // Compare displayed cards
    if (yourNumber > computerNumber)
    {
        // You win
        txtYouScore.Text =
Convert.ToString(Convert.ToInt32(txtYouScore.Text) + 2);
```

```
    }
    else if (computerNumber > yourNumber)
    {
        // Computer win
        txtCompScore.Text =
Convert.ToString(Convert.ToInt32(txtCompScore.Text) + 2);
    }
    else
    {
        // a tie!
        txtYouScore.Text =
Convert.ToString(Convert.ToInt32(txtYouScore.Text) + 1);
        txtCompScore.Text =
Convert.ToString(Convert.ToInt32(txtCompScore.Text) + 1);
    }
    cardIndex++;
    // Check to see if all cards have been used
    if (cardIndex > 25)
    {
        // Game over
        txtOver.Visible = true;
        btnNew.Text = "New Game";
        btnExit.Text = "Exit";
    }
}
```

You should be able to see each outlined step in this code. Notice particularly the shuffle routine and how **cardIndex** is used with the **cardNumber** array to display your card and the computer card. Remember the computer card is 26 elements ahead of your card in the cardNumber array. Look at how the card numbers are found and how comparisons are made. Check out the tricky way scores are updated without using any variables! Notice, too, how as your programming knowledge expands, there's a lot more happening in the code we write. Remember to use cut and paste where you can - it will make your work easier.

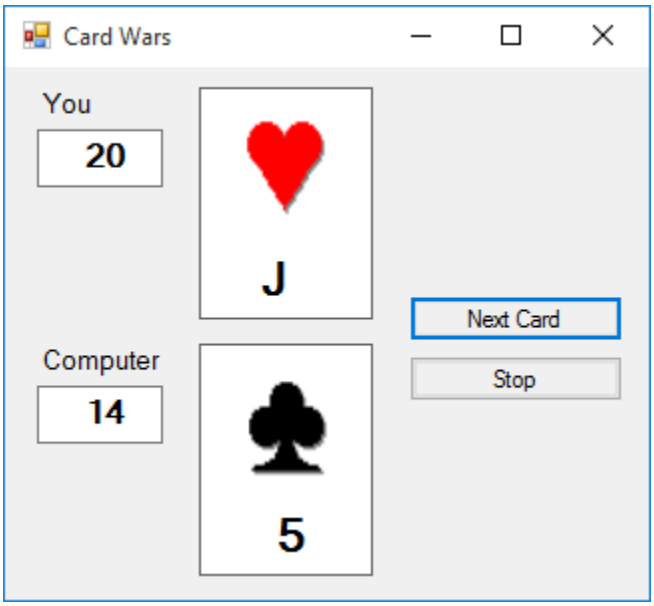
We now need to code the **btnExit_Click** event. Like **btnNew**, It also has two purposes. If the button **Text** property is **Exit**, the program stops. If the **Text** is **Stop**, the current game stops. That code is pretty simple:

```
private void btnExit_Click(object sender, EventArgs e)
{
    if (btnExit.Text == "Exit")
    {
        // Stop program
        this.Close();
    }
    else
    {
        //  Stop game
        txtOver.Visible = true;
        btnExit.Text = "Exit";
        btnNew.Text = "New Game";
    }
}
```

Save the project by clicking the **Save All** button in the toolbar.

Run the Project

Run the project. Click **New Game** to get started. Click **Next Card** to display each pair of cards. Notice how the different controls are used to make up the cards. Make sure the program works correctly. Here's what my screen looks like in the middle of a game:



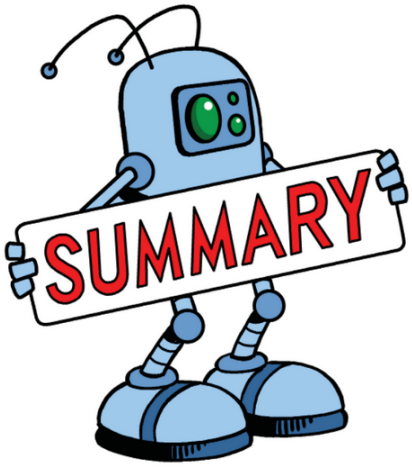
Play through one game and check each comparison to make sure you get the correct result and score with each new card. Make sure the **Stop** and **Exit** buttons work properly. Go through the usual process of making sure the program works as it should. Once you're convinced everything is OK, have fun playing the game. Share your creation with friends. If you made any changes during the running process, make sure you save the project.

Other Things to Try

Possible changes to the Card Wars project are obvious, but not easy. One change would be to have more than two players. Set up three and four player versions. You could also add a message after each comparison to way which player won (or whether it was a tie). You could use the txtOver control that's already there.

In Card Wars, we stop the game after going through the deck one time. In the real card game of War, after the first round, the players pick up the cards they won, shuffle them, and play another round. Every time a player uses all the cards in their "hand," they again pick up their winnings pile, reshuffle and continue playing. This continues until one player has lost all of their cards. Another change to Card Wars would be to write code that plays the game with these rules. As we said, it's not easy. You would need to add code to keep track of which cards each player won, when they ran out of cards to play, how to reshuffle their remaining cards, and new logic to see when a game was over. Such code would use more arrays, more for loops, and more variables. If you want a programming challenge, go for it!

And, while you're tackling challenges, here's another. In the usual War game, when two cards have the same value - War is declared! This means each player takes three cards from their "hand" and lays them face down. Then another card is placed face up. The higher card at that time wins all 10 cards! If it's still a tie, there's another War. Try adding this logic to the game. You might need to change the display to allow more cards. You'll need to figure out how to lay cards "face down" in C#. You'll need to check if a player has enough cards to wage War. Another difficult task, but give it a try if you feel adventurous.



This class presented one of the more challenging projects yet. The code involved in shuffling cards and displaying cards, though straightforward, was quite involved. The use of panel and picture box controls helped in the display. The use of arrays and for loops made the coding a bit easier. If you completely understood the Card Wars project, you are well on your way to being a good Visual C# programmer. Now, on to the last class.