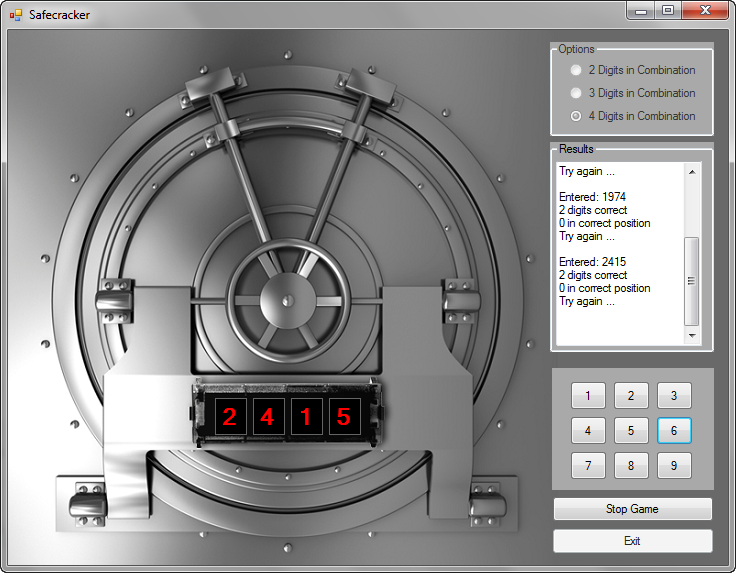
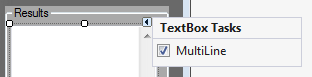
**Safecracker**

We will build the **Safecracker** game. A bank safe is locked and can only be opened if you enter the proper combination. The combination can be 2 to 4 non-repeating digits (digits range from 1 to 9). After each guess, you are told how many digits are correct and how many are in the correct location. Based on this information, you make another guess. You continue guessing until you get the correct combination or stop the game.



**DESIGN**

* The picture of the Vault is assigned to the Form’s **BACKGROUND Image** property. The **Background Image Layout** is set to None and the **BackColor** is set to DimGray.
* Labels are used to the combination display.
* A group box and radio buttons are used for selecting the number of digits in the combination.
* A group box with a textbox in it is used to display feedback to the user.
* Set the textbox’s multiline to true and set the scroll bar property to vertical.  
    
  
* Use a panel behind the 9 numbered buttons.

Run the project (press <**F5**>). The game will appear in its ‘stopped’ state, waiting for you to choose game options. The bank vault is disabled – no combination can be entered.

Click the **Start Game** button to start playing. Its caption will change (now reading **Stop Game**) and the **Options** group box and **Exit** button will become disabled. The panel control in the vault is now enabled showing a label control for each digit in the combination and activating the 9 button keypad. We call this the ‘playing’ state:

In this state, you make a guess using the keypad in the bank safe. After each guess, the results will be displayed in the text box in the **Results** group box.

Once a button is pressed, the corresponding digit appears in the proper label box. Also, the pressed button is disabled since the digits can’t repeat. Once three buttons are pressed, the results are shown. You are told how many digits in your guess were correct and how many were in the correct position.

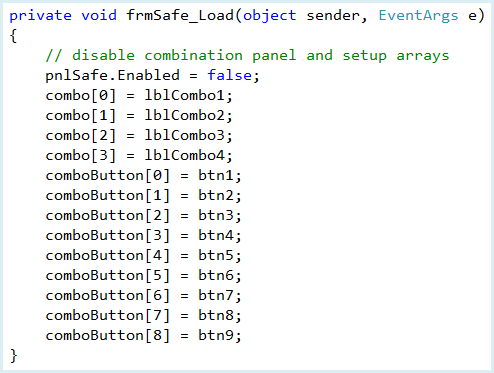
We will begin writing code for the game. We will write the code in several steps. As a first step, we will write the code that starts the game and establishes its ‘stopped’ state. Then, we look at how to go to ‘playing’ state following clicking of the **Start Game** button.

## Code Design – Initializing Stopped State

To make our programming job easier, we will establish control arrays for the four label controls used to display the combination and for the nine button controls used to enter a combination. These “form level” arrays are defined below the form constructor in the code window:

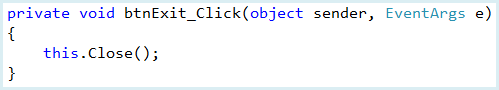
****

The assignments are made in the **frmSafecracker Load** event (we also disable the panel control holding the labels and buttons so no inputs can be made):



Run the project (press <**F5**>) to make sure the form is properly initialized. We have two choices at this point – either choose some options and click **btnStartStop** (the button with **Start Game**) or click **btnExit** (the button with **Exit**).

The code for exiting is simple. The **btnExit Click** event:



The code for the **btnStartStop** button is much more complicated. We will build it in several steps. First, we look at switching the game from stopped to playing state.

**Code Design – Stopped to Playing**

When the user clicks the **Start Game** button in ‘stopped’ state, several things must happen to switch the **Safecracker** game to ‘playing’ state:

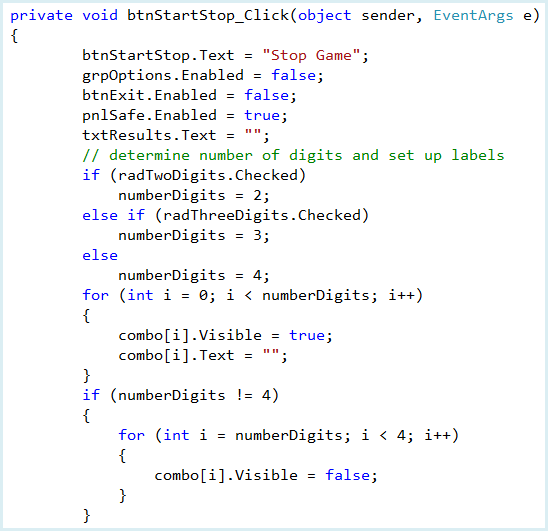
* Change the **Text** property of **btnStartStop** to **Stop Game**.
* Disable **grpOptions** (don’t allow selection of options while playing).
* Disable **btnExit**.
* Enable **pnlCombo** (allow entering a combination)
* Blank out **txtResults** (delete previous results).
* Display proper number of label controls to show combination.
* Determine secret combination.
* Allow user to input combination.
* Check to see if input combination matches secret combination – provide results.

Right now, we will implement each of these steps except those establishing the secret combination and accepting and checking user input. Those steps will be addressed separately.

Add this line to the form level declarations to define a variable for the number of digits in the combination:



The code for the **btnStartStop Click** event that implements all but the last three steps in the above method is:



**Q1:** What is the purpose of the code inside the final condition *if (numberDigits != 4)?*

Notice how the label control array (**combo**) is used to display the needed label controls for the displayed combination.

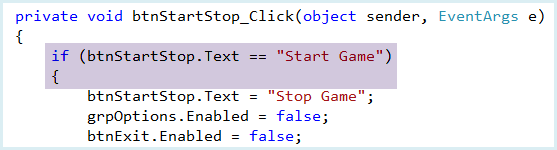
Save and run the project. Click **Start Game** and the game should switch to ‘playing’ state. Make sure the number of label boxes displayed in the vault area match the option selected. Notice the **Stop Game** button does nothing at the moment. Let’s fix that. Stop the project.

## Code Design – Playing to Stopped State

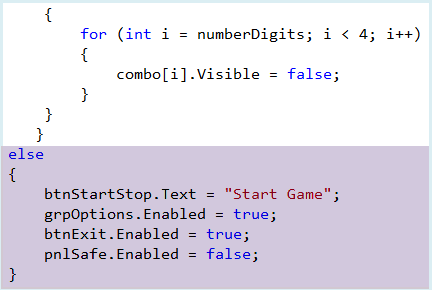
When the user clicks the **Stop Game** button in ‘playing’ state, several things must happen to switch the **Safecracker** game to ‘stopped’ state:

* Change the **Text** property of **btnStartStop** to **Start Game**.
* Enable **grpOptions**.
* Enable **btnExit**.
* Disable **pnlCombo.**

The button now marked **Stop Game** is the **btnStartStop** button. We have already added some code to its **Click** event. It is common practice to have one button control have multiple purposes - we just need to have some way to distinguish which “mode” the button is in when it is clicked. In this project, we use the Text property of the button. If the Text property is **Start Game**, we switch to ‘playing’ mode. If the Text property is **Stop Game**, we switch to ‘stopped’ mode. The code that does this is (modifications to the current **Click** event code are shaded):



**Code is continued below.**

****

Save and run the project. You should be able to now move from ‘stopped’ to ‘playing’ state and back. Make sure you can display 2, 3 or 4 digits in the combination. Try the **Exit** button. We can now look at the three missing steps in the program: generating a secret combination and accepting and checking user input.

## Code Design – Generating Secret Combination

A critical step in the **Safecracker** project is to generate the secret combination the game player is trying to guess. Using the options selected, we know the combination can be from 2 to 4 digits. We have also specified none of the digits can repeat.

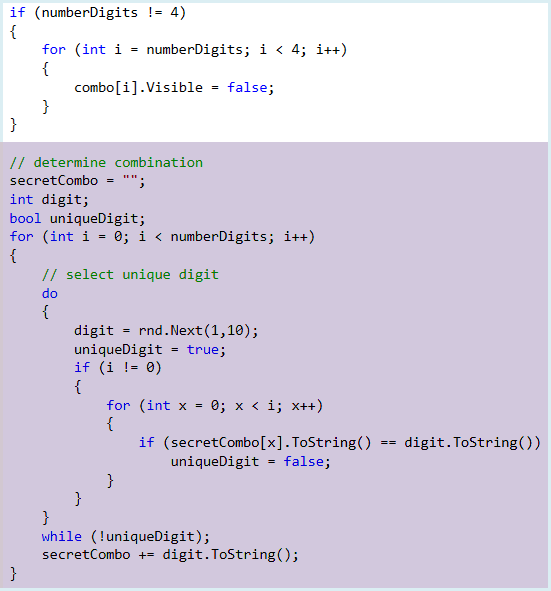
Add these form level variable declarations to define the secret combination and establish the random number object:

****

Since repeated digits are not allowed, for each digit in the combination, we generate a random number from 1 to 9, but only add it to the combination if it has not been used before. The code snippet that accomplishes this for our game is:

A string variable (**secretCombo**) is used to store the combination. It is initially blank. One digit at a time (a total of **numberDigits**) is added to **secretCombo** using string concatenation.

A random number **digit** (from 1 to 9) is generated. We then look at the current digits in the code to see if this newly generated digit is unique. If so, it is added to **secretCombo**. If not unique, a new random digit is generated. All of this logic is implemented in a **do** loop. Study how we examine the individual digits in **secretCombo** using an array The elements of this array are **char** type, so we need to convert to **String** before comparing. Add the given snippet to the **btnStartStop Click** event as shown in the shaded code below (we’ve also added a few new variable declarations.



**Q2:** Place a comment next to each line within the *do, while* loop.

We can now generate a secret combination, but have no way of seeing what it is. Let’s change that. Many times while developing a project, you want to check results before the project is complete. This means adding temporary code just for debugging purposes. Place this line after the closing bracket in the code generating the combination:

### Console.WriteLine(secretCombo);

This will write the generated combination in the **Output** window of the debugger allowing you to see if the combination meets the requirements (proper number of digits and proper repeating/non-repeating of digits).

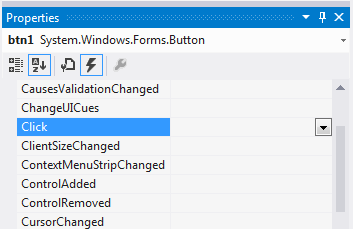
Save and run the project. Click **Start Game**. Open the **Output** window and look at the generated combination. Make sure it meets the selected options requirement. Click **Stop Game**. Repeat this process for various combinations to make sure the combination is being generated correctly. When done, delete the debug line from the code.

We’re now ready to accept the player’s guess at the secret combination using the 9 keypad buttons. We could write individual **Click** event methods for the 9 buttons to obtain the user’s input. Or, we can be smart and use one event method to handle clicking on any of the buttons. Let’s see how to do that.

## Handling Multiple Events in a Single Event Method

In most Visual C# Express projects, each event method handles a single event. When grouping controls, like the 9 keypad buttons here, it would be nice if a single method could handle multiple events. For example, if we have 4 buttons in a group, when one button is clicked, it would be preferable to have a single method where we decide which button was clicked, as opposed to having to monitor 4 separate event methods. Let’s see how to do this.

We use a button example to illustrate. Assume we have four buttons (**button1**, **button2**, **button3**, **button4**) that we want to share the same **Click** event method. To make this method sharing possible, we use the properties window to establish events. First, we select **button1**, go to the properties window and click the **Events** button.  The **Click** event will be highlighted since it is the default event for the button control:



Usually, at this point, we would double-click the event and the Visual C# environment would build an event method named **button1\_Click**. The name assigned to this method by Visual C# Express is arbitrary. We can assign any name we want and it would still handle the **Click** event for **button1**. To use a different name, we type the name next to the event in the properties window. Since the method will now handling multiple events, we would want to use a name to represent a group of buttons, not just a single button. A name like **myButtons\_Click** might be more appropriate – we will use that in this example.

Once you type a name for an event in the properties window and press <**Enter**>, the code window will open in the new method. For this example, we would see:

### private void myButtons\_Click(object sender, EventArgs e)

**{**

**}**

Right now, this method only handles the **Click** event for **button1**.

To have another control share this method, we again use the properties window. Next, we would choose **button2** in the properties window, then display its events and click the drop-down box that appears to the right of the **Click** event. One of the choices will be the **myButtons\_Click** event method. By simply choosing this existing method, it will now be the method called for the **Click** event for **button2**. We then repeat this process for each control we want to share the same event method. **Hint**: a quick way to do this is to “group select” every control on the form that is to share an event method. Then using event selection in the properties window, choose the shared method.

With this process, we can attach any existing event of any control to any method we want! It is best to append like events of like controls. It is possible to process dissimilar events of dissimilar controls with a single event method, but you need to be careful.

If we have a single method responding to events from multiple controls, how do we determine which particular event from which particular control invoked the method. How do we know which of the buttons is invoking the method? The **sender** argument of the event method provides the answer.

Each event method has a **sender** argument (the first argument of two) which identifies the control whose event caused the method to be invoked. With a little Visual C# coding, we can identify the **Name** property (or any other needed property) of the sending control. An example code is:

### Button btnExample; string buttonName;

**// Determine which button was clicked   
btnExample = (Button) sender;   
buttonName = btnExample.Name;**

In this code, we define a variable (**btnExample**) to be the type of the control attached to the method (a **Button** in this case). Then, we assign this variable to **sender** (after casting **sender** to the proper control type). Once this variable is established, we can determine any property we want to identify the button. In the above example, we find the button’s **Name** property. With this information, we now know which particular button was selected and we can process any code associated with this button. Notice a shortcut way to identify the **Name** property (without declaring any variables) is to use:

### ((Button) sender).Name

**Code Design – Accepting Player Input**

Once the secret combination has been generated, the next step is to allow the user to enter his or her guess at the combination. This is done using the 9 number buttons on the panel control in the bank graphic. The steps followed are:

* Click a button to enter a digit.
* If this is the first number in the combination, clear out any previous guess in label controls.
* Determine which button was clicked.
* Disable the clicked button since repeating digits are not allowed.
* Add the number on the clicked button to the user’s input combination.
* Display the number in the proper label control.
* If all digits have been entered, enable the number buttons and compare the user’s input to the actual combination, else repeat these steps.

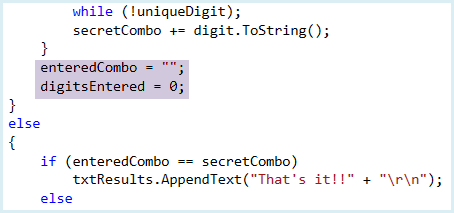
The code to perform these steps will be placed in a method named **btnCombo\_Click**. This method is set to handle the **Click** event for all 9 buttons used for input (**btn1**, **btn2**, **btn3**, **btn4**, **btn5**, **btn6**, **btn7**, **btn8**, **btn9**).

Add these form level variables to store the combination entered by the user and the number of digits entered:

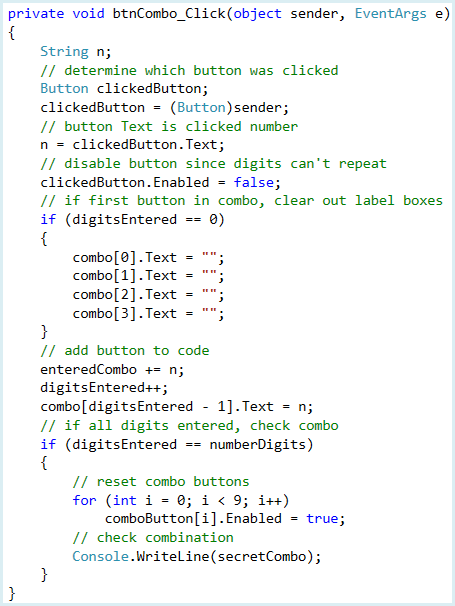
****

Initialize the value of these two new variables using the shaded code in

### btnStartStop\_Click:



Add this method (**btnCombo\_Click**) to your project. This implements the needed steps to check the user’s input:



**Q3:** Explain, in your own words, how this one method is able to take input from any of the number buttons.

Notice the **Text** property of the clicked button is the corresponding digit (the variable **n**). We have used a **WriteLine** line to print the complete entered combination in the **Output** window. Save and run the project. Make sure the entered combination looks correct (has the proper number of digits, in the correct order and following repeatability rules). Make sure the keypad buttons are re-enabled after entering a combination.

## Code Design – Checking Player Input

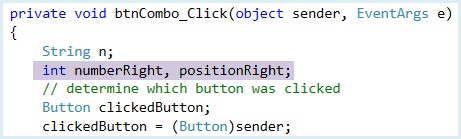
We now have the capability to generate a secret combination and obtain the player’s guess at that combination. We need the code that compares the two string variables and displays the results. If it’s a match, great! If not, we tell the player how many digits are correct and how many are in the correct position. This is some of the trickier code.

Once the user has completed their input, we follow these steps:

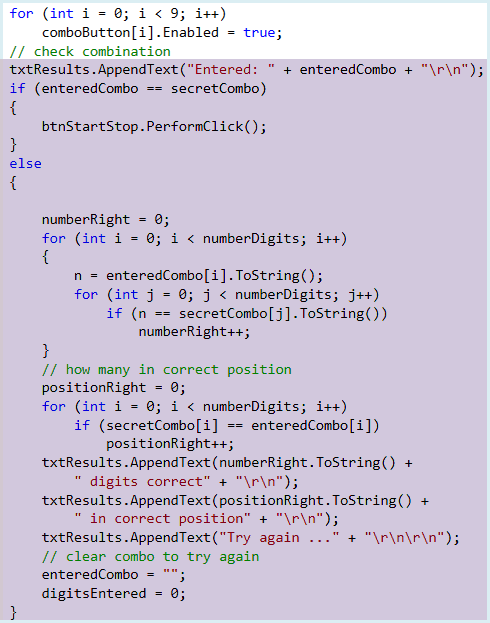
* Display the user’s guess (**enteredCombo**) in **txtResults**.
* If user’s guess (**enteredCombo**) equals the actual combination (**secretCombo**), return the game to stopped state and let the user know they guessed the combination.
* If incorrect guess, determine number of correct digits and number in correct positions. Display this information in **txtResults**.
* Clear user’s guess to allow another try.

All of the code to perform these steps is placed in the **btnCombo Click** method.

Place the shaded code in the **btnCombo Click** method to implement these steps (we deleted the **WriteLine** line):



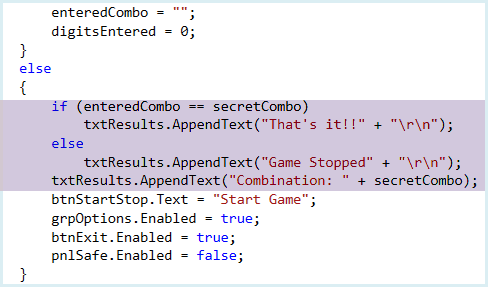
**Code is continued below.**



**Q4:** Explain the difference between *numberRight* and *positionRight,* and how the program decides to increment each one.

You should be able to follow the above code. It first counts the number of correct digits in the user’s input. The code to see if digits are in correct position just compares each digit in **enteredCombo** and **secretCombo**. All the results are displayed in the **txtResults** text box using the **AppendText** method.

If the correct combination is entered, we want to return to ‘stopped’ state. An easy way to do this is simply programmatically click on the **Stop Game** button using the **PerformClick** method. Using this approach, though, we need to add some code to the **btnStartStop Click** event. The new code is used to distinguish between stopping the game before guessing the combination and stopping because the combination has been guessed. Make the shaded changes to the **btnStartStop Click** method:



Save and run the project. The game is now fully functional. Try playing it a few times to make sure things are working okay.

**Level 4+ Achievements**

* After each guess, you might like to provide a hint on how to proceed.
* After a correct combination, maybe have more dramatic results. Perhaps have the vault graphic disappear and a pile of treasure appear.
* Count the number of guesses and display high scores.