

6

UpDown Control, Decisions, Random Numbers



Review and Preview

You're halfway through the course! You should now feel comfortable with the project building process and the controls you've studied. In the rest of the classes, we will concentrate more on controls and C# and less on the Visual C# environment. In this class, we look at the numeric updown control, at decisions using C#, and at a very fun topic, the random number. You will build a 'Guess the Number' game project.

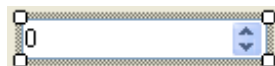
Numeric UpDown Control

The **Numeric UpDown** control is used to obtain a numeric input. It looks like a text box control with two small arrows. Clicking the arrows changes the displayed value, which ranges from a specified minimum to a specified maximum. The user can even type in a value, if desired. These controls are useful for supplying an integer number, such as a date in a month. The numeric updown control is selected from the Visual C# toolbox. It appears as:

In Toolbox:



On Form (default properties):



Properties

The numeric updown properties are:

<u>Property</u>	<u>Description</u>
Name	Name used to identify numeric updown control. Three letter prefix for numeric updown name is nud .
Value	Value assigned to the updown control (a decimal type).
Increment	Amount to increment (increase) or decrement (decrease) the updown control when the up or down buttons are clicked.
Maximum	Maximum value for the updown control.
Minimum	Minimum value for the updown control.
TextAlign	Sets whether displayed value is left-justified, right-justified or centered.

Font	Sets style, size, and type of displayed value text.
BackColor	Sets updown control background color.
ForeColor	Sets of color of displayed value text.
Left	Distance from left side of form to left side of updown control (X in properties window, expand Location property).
Top	Distance from top side of form to top side of updown control (Y in properties window, expand Location property).
Width	Width of the updown control in pixels (expand Size property).
Height	Height of updown control in pixels (expand Size property).
ReadOnly	Determines whether the text may be changed by the use of the up or down buttons only.
Enabled	Determines whether updown control can respond to user events (in run mode).
Visible	Determines whether the updown control appears on the form (in run mode).

Operation of the numeric updown control is actually quite simple. The **Value** property can be changed by clicking either of the arrows (value will be changed by **Increment** with each click) or, optionally by typing a value (if **ReadOnly** is **False**). If using the arrows, the value will always lie between **Minimum** and **Maximum**. If the user can type in a value, you have no control over what value is typed.

Example

Start Visual C# and start a new project. We will create a numeric updown control that provides numbers from 0 to 20. Put a numeric updown control on the form. (Make sure you choose the **numeric updown** control and not the domain updown control.) Resize it and move it, if desired. Set the following properties:

Property	Value
Value	10
Increment	1
Minimum	0
Maximum	20
ReadOnly	False

If you like, try changing colors, font and any other properties too. This numeric updown control has an initial Value of 10. The smallest Value can be is 0 (Minimum), the largest it can be is 20 (Maximum). Value will change by 1 (Increment) when an arrow is clicked. The user can type a value (ReadOnly is False) if desired.

Run the project. The numeric updown control will appear and display a value of 10. Click the end arrows and see the value change. Notice the value will not drop below 0 or above 20, the limits established at design time. Click the display area and type in a value of 100. Note that, even though this is higher than the maximum of 20, you can type the value. Try increasing the value, it will not increase. Hit **<Enter>** or try to decrease the value and it is immediately adjusted within the limits you set. So, Visual C# makes some attempts to make sure the user doesn't type illegal values. Stop the project.

Events

We will only use a single numeric updown event:

<u>Event</u>	<u>Description</u>
ValueChanged	Occurs when the Value property has been changed in some way.

The **ValueChanged** event is executed whenever Value changes. This is where you can use the current value. If the user is allowed to type a value in the control, you might have to check if it is within acceptable limits.

Typical Use of Numeric UpDown Control

The usual design steps for a numeric updown control are:

- Set the **Name**, **Minimum** and **Maximum** properties. Initialize **Value** property. Decide on value for **ReadOnly**.
- Monitor **ValueChanged** event for changes in Value.
- You may also want to change the **Font**, **BackColor** and **ForeColor** properties.

Note, for maximum flexibility, **Value** is a **decimal** type, meaning you can have non-integer numbers (you will need to set the **DecimalPlaces** property to some value). If you want an integer value, you need to cast **Value** to an **int** type.

C# - The Third Lesson

In the C# lesson for this class, we learn about one of the more useful functions of a computer program - decision making. We will discuss expressions and operators used in decisions and how decisions can be made. We will also look at a new C# function - the random number. This function is the heart of every computer game.

Logical Expressions

You may think that computers are quite smart. They appear to have the ability to make amazing decisions and choices. Computers can beat masters at chess and help put men and women into space. Well, computers really aren't that smart - the only decision making ability they have is to tell if something is **true** or **false**. But, computers have the ability to make such decisions very quickly and that's why they appear smart (and because, unlike the True or False tests you take in school, computers always get the right answer!). To use C# for decision making, we write all possible decisions in the form of **true or false?** statements, called **logical expressions**. We give the computer a logical expression and the computer will tell us if that expression is true or false. Based on that decision, we can take whatever action we want in our computer program. Note the result of a logical expression is a **bool** type value.

Say in a computer program we need to know if the value of the variable **aValue** is larger than the value of the variable **bValue**. We would ask the computer (by writing some C# code) to provide an answer to the true or false? statement: "aValue is larger than bValue." This is an example of a logical expression. If the computer told us this was true, we could take one set of C# steps. If it was false, we could take another. This is how decisions are done in C#.

To make decisions, we need to know how to build and use logical expressions.
The first step in building such expressions is to learn about comparison operators.

Comparison Operators

In the Class 3, we looked at one type of C# operator - arithmetic operators. In this class, we introduce the idea of a **comparison operator**. Comparison operators do exactly what they say - they compare two values, with the output of the comparison being a Boolean value (**bool**). That is, the result of the comparison is either **true** or **false**. Comparison operators allow us to construct logical expressions that can be used in decision making.

There are six comparison operators. The first is the “**equal to**” operator represented by two equal (==) signs. This operator tells us if two values are equal to each other. Examples are:

Comparison	Result
6 == 7	false
4 == 4	true

A common error (a logic error) in C# is to only use one equal sign for the “equal to” operator. Using a single equal sign simply assigns a value to the variable making it always true

There is also a “**not equal to**” operator represented by a symbol consisting of an exclamation point (called the **not** operator) followed by the equal sign (**!=**).

Examples of using this operator:

Comparison	Result
6 != 7	true
4 != 4	false

There are other operators that let us compare the size of numbers. The “**greater than**” operator (**>**) tells us if one number (left side of operator) is greater than another (right side of operator). Examples of its usage:

Comparison	Result
8 > 3	true
6 > 7	false
4 > 4	false

The “**less than**” operator (**<**) tells us if one number (left side of operator) is less than another (right side of operator). Some examples are:

Comparison	Result
8 < 3	false
6 < 7	true
4 < 4	false

The last two operators are modifications to the “greater than” and “less than” operators. The “**greater than or equal to**” operator (**>=**) compares two numbers. The result is true if the number on the left of the operator is greater than or equal to the number on the right. Otherwise, the result is false. Examples:

Comparison	Result
8 >= 3	true
6 >= 7	false
4 >= 4	true

Similarly, the “**less than or equal to**” operator (**<=**) tells us if one number (left side of operator) is less than or equal to another (right side of operator). Examples:

Comparison	Result
8 <= 3	false
6 <= 7	true
4 <= 4	true

Comparison operators have equal precedence among themselves, but are lower than the precedence of arithmetic operators. This means comparisons are done after any arithmetic. Comparison operators allow us to make single decisions about the relative size of values and variables. What if we need to make multiple decisions? For example, what if we want to know if a particular variable is smaller than one number, but larger than another? We need ways to combine logical expressions - logical operators can do this.

Logical Operators

Logical operators are used to combine logical expressions built using comparison operators. Using such operators allows you, as the programmer, to make any decision you want. As an example, say you need to know if two variables named **aValue** and **bValue** are both greater than 0. Using the “greater than” comparison operator (**>**), we know how to see if aValue is greater than zero and we know how to check if bValue is greater than 0, but how do we combine these expressions and obtain one Boolean result (true or false)?

We will look at two logical operators used to combine logical expressions. The first is the **and** operator represented by two ampersands (**&&**). The format for using this operator is (using two logical expressions, **x** and **y**, each with a Boolean (**bool** type) result):

x && y

This expression is asking the question “are x and y both true?” That’s why it is called the and operator. The and operator (**&&**) will return a true value only if both x and y are true. If either expression is false, the and operator will return a false. The four possibilities for **and** (**&&**) are shown in this **logic table**:

x	y	x && y
true	true	true
true	false	false
false	true	false
false	false	false

Notice the **and** operator would be used to solve the problem mentioned in the beginning of this section. That is, to see if the variables `aValue` and `bValue` are both greater than zero, we would use the expression:

```
aValue > 0 && bValue > 0
```

The other logical operator we will use is the **or** operator represented by two pipes (`||`). The pipe symbol is the shift of the backslash key (`\`) on a standard keyboard. The format for using this operator is:

```
x || y
```

This expression is asking the question “is x or y true?” That’s why it is called the or operator. The or (`||`) operator will return a true value if either x or y is true. If both expressions are false, the or operator will return a false. The four possibilities for **or** (`||`) are:

x	y	x y
true	true	true
true	false	true
false	true	true
false	false	false

The or operator is second in precedence to the and operator (that is, **and** is done before **or**), and all logical operators come after the comparison operators in precedence. Use of comparison operators and logical operators to form logical expressions is key to making proper decisions in C#. Make sure you understand how all the operators (and their precedence) work. Let's look at some examples to help in this understanding.

In these examples, we will have two integer variables **aInteger** and **bInteger**, with values:

```
aInteger = 14  
bInteger = 7
```

What if we want to evaluate the logical expression:

```
aInteger > 10 && bInteger > 10
```

Comparisons are done first, left to right since all comparison operators share the same level of precedence. **aInteger** (14) is greater than 10, so **aInteger > 10** is true. **bInteger** (7) is not greater than 10, so **bInteger > 10** is false. Since one expression is not true, the result of the and (&&) operation is false. This expression '**aInteger > 10 && bInteger > 10**' is false. What is the result of this expression:

```
aInteger > 10 || bInteger > 10
```

Can you see this expression is true (**aInteger > 10** is true, **bInteger > 10** is false; true || false is true)?

There is no requirement that a logical expression have just one logical operator. So, let's complicate things a bit. What if the expression is:

```
aInteger > 10 || bInteger > 10 && aInteger + bInteger == 20
```

Precedence tells us the arithmetic is done first (aInteger and bInteger are added), then the comparisons, left to right. We know aInteger > 10 is true, bInteger > 10 is false, aInteger + bInteger == 20 is false. So, this expression, in terms of boolean comparison values, becomes:

```
true || false && false
```

How do we evaluate this? Precedence says the and (&&) is done first, then the or (||). The result of 'false && false' is false, so the expression reduces to:

```
true || false
```

which has a result of true. Hence, we say the expression 'aInteger > 10 || B > 10 && aInteger + bInteger = 20' is true.

Parentheses can be used in logical expressions to force precedence in evaluations. What if, in the above example, we wanted to do the or (||) operation first? This is done by rewriting using parentheses:

```
(aInteger > 10 || bInteger > 10) && aInteger + bInteger == 20
```

You should be able to show this evaluates to false [do the or (||) first]. Before, without parentheses, it was true. The addition of parentheses has changed the value of this logical expression! It's always best to clearly indicate how you want a logical expression to be evaluated. Parentheses are a good way to do this. Use parentheses even if precedence is not affected.

If we moved the parentheses in this example and wrote:

```
aInteger > 10 || (bInteger > 10 && aInteger + bInteger == 20)
```

the result (true) is the same as if the parentheses were not there since the and (&&) is done first anyway. The parentheses do, however, clearly indicate the and is performed first. Such clarity is good in programming.

Comparison and logical operators are keys to making decisions in C#. Make sure you are comfortable with their meaning and use. Always double-check any logical expression you form to make sure it truly represents the decision logic you intend. Use parentheses to add clarity, if needed.

Decisions - The if Statement

We've spent a lot of time covering comparison operators and logical operators and discussed how they are used to form logical expressions. But, just how is all this used in computer decision making? We'll address that now by looking at the C# **if** statement. Actually, the if statement is not a single statement, but rather a group of statements that implements some decision logic. It is conceptually simple.

The if statement checks a particular logical expression with a Boolean (**bool** type) result. It executes different groups of C# statements, depending on whether that expression is true or false. The C# structure for this logic is:

```
if (expression)
{
    [C# code block to be executed if expression is true]
}
else
{
    [C# code block to be executed if expression is false]
}
```

Let's see what goes on here. We have some logical **expression** which is formed from comparison operators and logical operators. **if** expression is true, then the first block of C# statements (marked by a pair of left and right curly braces) is executed. **else** (meaning expression is not true, or it is false), the second block of C# statements is executed. Each block of code contains standard C# statements, indented by some amount. Whether expression is true or false, program execution continues with the first line of C# code after the last right curly brace (**}**).

The `else` keyword and the block of statements following the `else` are optional. If there is no C# code to be executed if expression is false, the if structure would simply be:

```
if (expression)
{
    [C# code block to be executed if expression is true]
}
```

Let's try some examples.

The neighborhood kids just opened a lemonade stand and you want to let the computer decide how much they should charge for each cup sold. Define an **int** type variable **cost** (cost per cup in cents - our foreign friends can use some other unit here) and another **int** variable **temperature** (outside temperature in degrees F - our foreign friends would, of course, use degrees C). We will write an if structure that implements a decision process that establishes a value for cost, depending on the value of temperature. Look at the C# code:

```
if (temperature > 90)
{
    cost = 50;
}
else
{
    cost = 25;
}
```

We see that if `temperature > 90` (a warm day, hence we can charge more), a logical expression, is true, the cost will be 50, else (meaning temperature is not greater than 90) the cost will be 25. Not too difficult. Notice that we have indented the lines of C# code in the two blocks (one line of code in each block here). This is common practice in writing C# code. It clearly indicates what is done in each

case and allows us to see where an if structure begins and ends. The Visual C# environment will actually handle the indenting for you.

We could rewrite this (and get the same result) without the else statement. Notice, this code is equivalent to the above code:

```
cost = 25;  
if (temperature > 90)  
{  
    cost = 50  
}
```

Here, before the if structure, cost is 25. Only if temperature is greater than 90 is cost changed to 50. Otherwise, cost remains at 25. Even though, in these examples, we only have one line of C# code that is executed for each decision possibility, we are not limited to a single line. We may have as many lines of C# code as needed in the code blocks of if structures.

What if, in our lemonade stand example, we want to divide our pricing structure into several different cost values, based on several different temperature values? The if structure can be modified to include an **else if** statement to consider multiple logical expressions. Such a structure is:

```
if (expression1)
{
    [C# code block to be executed if expression1 is true]
}
else if (expression2)
{
    [C# code block to be executed if expression2 is true]
}
else if (expression3)
{
    [C# code block to be executed if expression3 is true]
}
else
{
    [C# code block to be executed if expression1, expression 2, and
    expression3 are all false]
}
```

Can you see what happens here? It's pretty straightforward - just work down through the code. If expression1 is true, the first block of C# code is executed. If expression1 is false, the program checks to see if expression2 (using the else if) is true. If expression2 is true, that block of code is executed. If expression2 is false, expression3 is evaluated. If expression3 is true, the corresponding code block is executed. If expression3 is false, and note by this time, expression1, expression2, and expression3 have all been found to be false, the code in the else block (and this is optional) is executed.

You can have as many else if statements as you want. You must realize, however, that only one block of C# code in an if structure will be executed. This means that once C# has found a logical expression that is true, it will execute that block of code then leave the structure and execute the first line of code following

the last right curly brace (}). For example, if in the above example, both expression1 and expression3 are true, only the C# statements associated with expression1 being true will be executed. The rule for if structures is: only the code block associated with the first true expression will be executed.

How can we use this in our lemonade example? A more detailed pricing structure is reflected in this code:

```
if (temperature > 90)
{
    cost = 50;
}
else if (temperature > 80)
{
    cost = 40;
}
else if (temperature > 70)
{
    cost = 30;
}
Else
{
    cost = 25;
}
```

What would the cost be if temperature is 85? temperature is not greater than 90, but is greater than 80, so cost is 40.

What if this code was rewritten as:

```
if (temperature > 70)
{
    cost = 30;
}
else if (temperature > 80)
{
    cost = 40;
}
else if (temperature > 90)
{
    cost = 50;
}
Else
{
    cost = 25;
}
```

This doesn't look that different - we've just reordered some statements. But, notice what happens if we try to find cost for temperature equal to 85 again. The first if expression is true (temperature is greater than 70), so cost is 30. This is not the result we wanted and will decrease profits for our lemonade stand! Here's a case where the "first true" rule gave us an incorrect answer - a logic error.

This example points out the necessity to always carefully check any if structures you write. Make sure the decision logic you want to implement is working properly. Make sure you try cases that execute all possible decisions and that you get the correct results. The examples used here are relatively simple. Obviously, the if structure can be more far more complicated. Using multiple variables, multiple comparisons and multiple operators, you can develop very detailed decision making processes. In the remaining class projects, you will see examples of such processes.

Random Number Generator

Let's leave decisions for now and look at a fun C# concept - the random number. Have you ever played the Windows solitaire card game or Minesweeper or some similar game? Did you notice that every time you play the game, you get different results? How does this happen? How can you make a computer program unpredictable or introduce the idea of "randomness?" The key is the C# random number generator. This generator simply produces a different number every time it is referenced.

Why do you need random numbers? In the Windows solitaire card game, the computer needs to shuffle a deck of cards. It needs to "randomly" sort fifty-two cards. It uses random numbers to do this. If you have a game that rolls a die, you need to randomly generate a number between 1 and 6. Random numbers can be used to do this. If you need to flip a coin, you need to generate Heads or Tails randomly. Yes, random numbers are used to do this too.

Visual C# has several methods for generating random numbers. We will use just one of them – a random generator of integers (whole numbers). The generator uses what is called the **Random** object. Don't worry too much about what this means –just think of it as another variable type. Follow these few steps to use it. First create a **Random** object (we'll name it **myRandom**) using the **constructor**:

```
Random myRandom = new Random();
```

This statement is placed with the variable declaration statements.

Now, whenever you need a random integer value, use the **Next** method of this Random object we created:

```
myRandom.Next(limit)
```

This statement generates a random integer value that is greater than or equal to 0 and less than **limit**. Note it is less than limit, not equal to. For example, the method:

```
myRandom.Next(5)
```

will generate random numbers from 0 to 4. The possible values will be 0, 1, 2, 3 and 4.

Let's try it. Start Visual C# and start a new project. Put a button (default name **button1**) and label control (default name **label1**) on the form. We won't worry about properties or names here - we're just playing around, not building a real project. In fact, that's one neat thing about Visual C#, it is easy to play around with. Open the code window and add this line under the form constructor to create the random number object:

```
Random myRandom = new Random();
```

Then, put this code in the **button1_Click** event method:

```
private void button1_Click(object sender, EventArgs e)
{
    label1.Text = Convert.ToString(myRandom.Next(10));
}
```

This code simply generates a random integer between 0 and 9 (**Next** uses a limit of 10) and displays its value in the label control (after converting it to a string type). Run the project. Click the button. A number should appear in the label control.

Click the button again and again. Notice the displayed number changes with each click and there is no predictability to the number - it is random. The number printed should always be between 0 and 9. Try other limit values if you'd like to understand how the random object works.

So, the random number generator object can be used to introduce randomness in a project. This opens up a lot of possibilities to you as a programmer. Every computer game, video game, and computer simulation, like sports games and flight simulators, use random numbers. A roll of a die can produce a number from 1 to 6. To use our **myRandom** object to roll a die, we would write:

```
dieNumber = myRandom.Next(6) + 1;
```

For a deck of cards, the random integers would range from 1 to 52 since there are 52 cards in a standard playing deck. Code to do this:

```
cardNumber = myRandom.Next(52) + 1;
```

If we want a number between 0 and 100, we would use:

```
yourNumber = myRandom.Next(101);
```

Check the examples above to make sure you see how the random number generator produces the desired range of integers. Now, let's move on to a project that will use this generator.

Project - Guess the Number Game

Back in the early 1980's, the first computers intended for home use appeared. Brands like Atari, Coleco, Texas Instruments, and Commodore were sold in stores like Sears and Toys R Us (sorry, I can't type the needed 'backwards' R). These computers didn't have much memory, couldn't do real fancy graphics, and, compared to today's computers, cost a lot of money. But, these computers introduced a lot of people to the world of computer programming. Many games (usually written in the BASIC language) appeared at that time and the project you will build here is one of those classics.

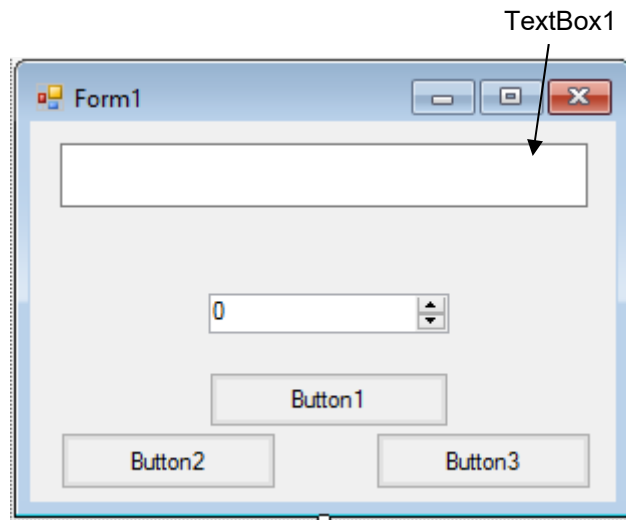
Project Design

You've all played the game where someone said "I'm thinking of a number between 1 and 10" (or some other limits). Then, you try to guess the number. The person thinking of the number tells you if you're low or high and you guess again. You continue guessing until you finally guess the number they were thinking of. We will develop a computer version of this game here. The computer will pick a number between 0 and 100 (using the random number generator). You will try to guess the number. Based on your guess, the computer will tell you if you are Too Low or Too High.

Several controls will be needed. Buttons will control game play (one to tell the computer to pick a number, one to tell the computer to check your guess, and one to exit the program). We will use a numeric updown control to set and display your guess. A label control will display the computer's messages to you. This project is saved as **GuessNumber** in the course projects folder (**\BeginVCS\BVCS Projects**).

Place Controls on Form

Start a new project in Visual C#. Place three buttons, a text box, and a numeric updown control on the form. Move and size controls until your form should look something like this:



Set Control Properties

Set the control properties using the properties window:

Form1 Form:

Property Name	Property Value
Text	Guess the Number
FormBorderStyle	Fixed Single
StartPosition	CenterScreen

textBox1 Text Box:

Property Name	Property Value
Name	txtMessage
TextAlign	Center
Font	Arial
Font Size	16
BackColor	White
ForeColor	Blue
ReadOnly	True
TabStop	False

numericUpDown1 Numeric UpDown:

Property Name	Property Value
Name	nudGuess
Font	Arial
Font Size	16
BackColor	White
ForeColor	Red
TextAlign	Center
Value	50
Minimum	0
Maximum	100
Increment	1
ReadOnly	True
Enabled	False

button1 Button:

Property Name	Property Value
Name	btnCheck
Text	Check Guess
Enabled	False

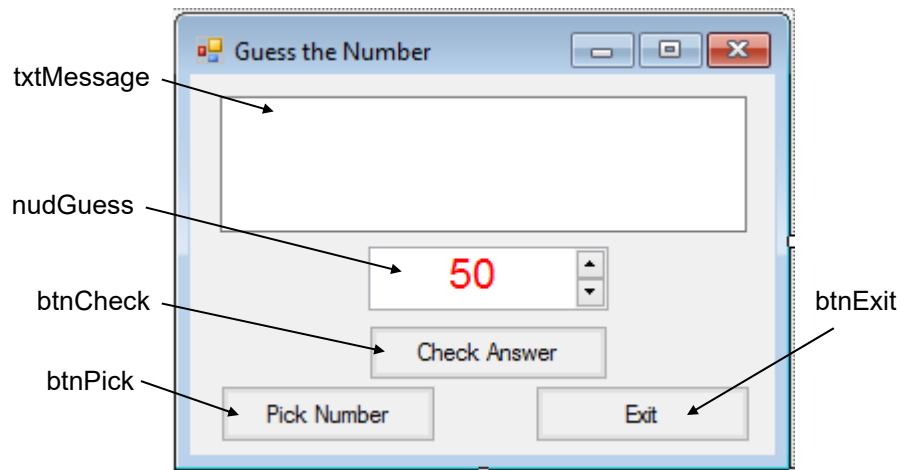
button2 Button:

Property Name	Property Value
Name	btnPick
Text	Pick Number

button3 Button:

Property Name	Property Value
Name	btnExit
Text	Exit

When done, your form should look something like this (you may have to move and resize a few controls around to get things to fit):



We have set the **Enabled** properties of **btnCheck** and **nudGuess** to **False** initially. We do not want to allow guesses until the **Pick Number** button is clicked.

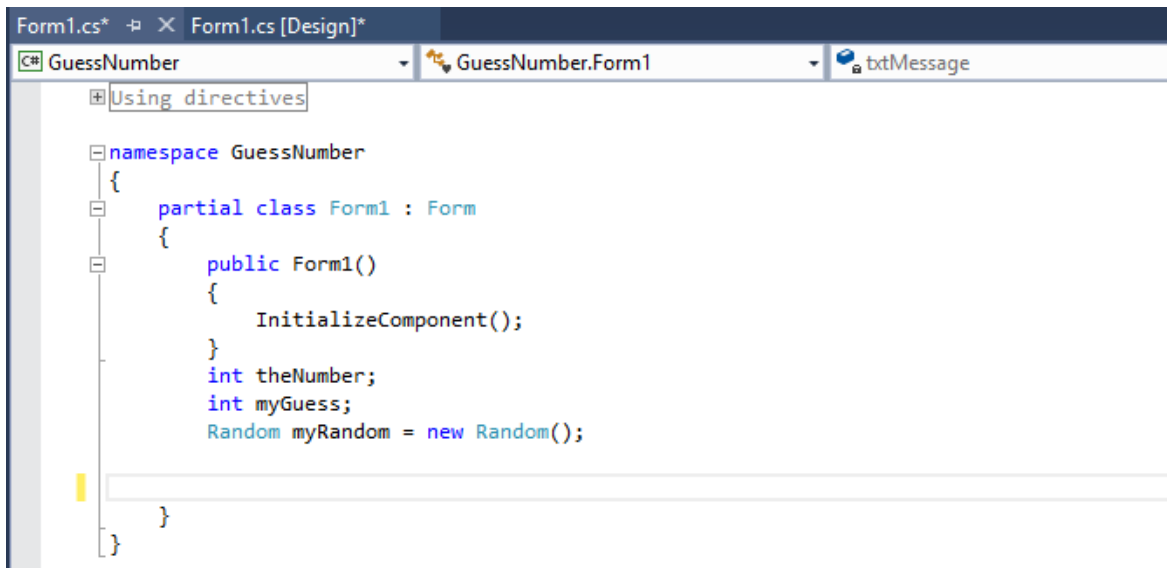
Write Event Methods

How does this project work? You click **Pick Number** to have the computer pick a number to guess. This click event will 'enable' the numeric updown control and **Check Guess** button (remember we set their Enabled properties initially at False). Input your guess using the numeric updown control, then click Check Guess. The computer will tell you if your guess is too low, too high, or correct by using the message box (**txtMessage**). So, we need a **Click** event method for each button.

Two variables are needed in this project, both **int** types. One variable will store the number selected by the computer (the number you are trying to guess). We call this variable **theNumber**. Your current guess will be saved in the variable **myGuess**. You will also need a Random object named **myRandom**. Open the code window and declare these variables in the **general declarations** area under the form constructor method:

```
int theNumber;  
int myGuess;  
Random myRandom = new Random();
```

After typing these lines, the code window should appear as:



```
Form1.cs*  Form1.cs [Design]*
C# GuessNumber  GuessNumber.Form1  txtMessage
+ Using directives
- namespace GuessNumber
{
-   partial class Form1 : Form
    {
-       public Form1()
        {
            InitializeComponent();
        }
        int theNumber;
        int myGuess;
        Random myRandom = new Random();
    }
}
```

When you click **Pick Number**, the computer needs to perform the following steps:

- Pick a random integer number between 0 and 100.
- Display a message to the user.
- Enable the numeric updown control to allow guesses.
- Enable the Check Guess button to allow guesses.

And, we will add one more step. Many times in Visual C#, you might want to change the function of a particular control while the program is running. In this game, once we click Pick Number, that button has no further use until the number has been guessed. But, we need a button to tell us the answer if we choose to give up before guessing it. We will use **btnPick** to do this. We will change the Text property and add decision logic to see which 'state' the button is in. If the button says "Pick Number" when it is clicked (the initial state), the above steps will be followed. If the button says "Show Answer" when it is clicked (the 'playing' state), the answer will be shown and the form controls returned to their initial state. This is a common thing to do in Visual C#.

Here's the **btnPick_Click** code that does everything:

```
private void btnPick_Click(object sender, EventArgs e)
{
    if (btnPick.Text == "Pick Number")
    {
        // Get new number and set controls
        theNumber = myRandom.Next(101);
        txtMessage.Text = "I'm thinking of a number between 0
and 100";
        nudGuess.Value = 50;
        nudGuess.Enabled = true;
        btnCheck.Enabled = true;
        btnPick.Text = "Show Answer";
    }
    else
    {
        // Just show the answer and re-set controls
        txtMessage.Text = "The answer is" +
Convert.ToString(theNumber);
        nudGuess.Value = theNumber;
        nudGuess.Enabled = false;
        btnCheck.Enabled = false;
        btnPick.Text = "Pick Number";
    }
}
```

Study this so you see what is going on. Notice the use of indentation in the if structure. Notice in the lines where we set the `txtMessage.Text`, it looks like two lines of C# code. Type this all on one line - the word processor is making it look like two. In fact, keep an eye out for such things in these notes. It's obvious where a so-called "word wrap" occurs.

When you click **Check Answer**, the computer should see if your current guess (myGuess) is correct (the **Value** returned by the numeric updown control needs to be converted to an **int** type before doing the comparison with myGuess). If so, a message telling you so will appear and the form controls return to their initial state, ready for another game. If not, the computer will display a message telling you if you are too low or too high. You can then make another guess. The **btnCheck_Click** event that implements this logic is:

```
private void btnCheck_Click(object sender, EventArgs e)
{
    // Guess is the updown control value
    myGuess = (int) nudGuess.Value;
    if (myGuess == theNumber)
    {
        // Correct guess
        txtMessage.Text = "That's it!!";
        nudGuess.Enabled = false;
        btnCheck.Enabled = false;
        btnPick.Text = "Pick Number";
    }
    else if (myGuess < theNumber)
    {
        // Guess is too low
        txtMessage.Text = "Too low!";
    }
    else
    {
        // Guess is too high
        txtMessage.Text = "Too high!";
    }
}
```

The last button click event is **btnExit_Click**:

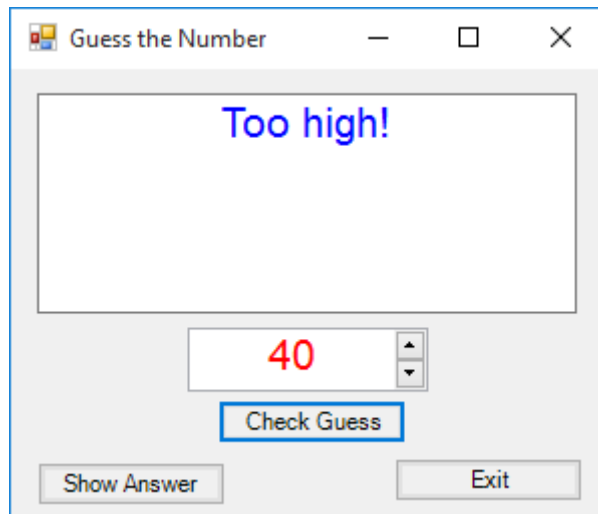
```
private void btnExit_Click(object sender, EventArgs e)
{
    this.Close();
}
```

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

Run the Project

Run the project. Click **Pick Number** to have the computer to pick a number to guess. Use the arrows on the numeric updown control to input your guess. Click **Check Guess**. Continue adjusting your guess (using the computer clues) until you get the correct answer. Make sure the proper messages display at the proper times. Do you see how the text displayed on btnPick changes as the game 'state' changes? Make sure the **Show Answer** button works properly. Again, always thoroughly test your project to make sure all options work. Save your project if you needed to make any changes.

Here's what the form should look like in the middle of a game:



Other Things to Try

You can add other features to this game. One suggestion is to add a text box where the user can input the upper range of numbers that can be guessed. That way, the game could be played by a wide variety of players. Use a maximum value of 10 for little kids, 1000 for older kids.

Another good modification would be to offer more informative messages following a guess. Have you ever played the game where you try to find something and the person who hid the item tells you, as you move around the room, that you are freezing (far away), cold (closer), warm (closer yet), hot (very close), or burning up (right on top of the hidden item)? Try to modify the Guess the Number game to give these kind of clues. That is, the closer you are to the correct number, the warmer you get. To make this change, you will probably need the C# **absolute value** method, **Math.Abs**. This function returns the value of a number while ignoring its sign (positive or negative). The format for using Math.Abs is:

```
yourValue = Math.Abs(inputValue);
```

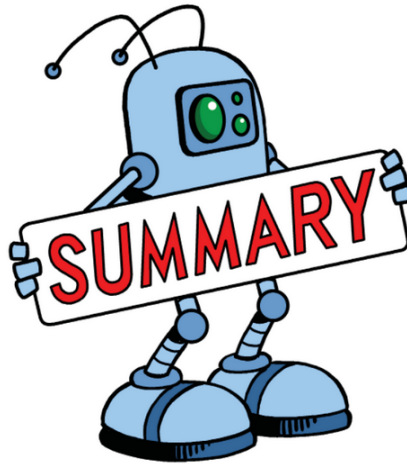
If inputValue is a positive number (greater than zero), yourValue is assigned inputValue. If inputValue is a negative number (less than zero), yourValue is assigned the numerical value of inputValue, without the minus sign. A few examples:

value	Math.Abs(value)
6	6
-6	6
0	0
-1.1	1.1

In our number guessing game, we can use `Math.Abs` to see how close a guess is to the actual number. One possible decision logic is:

```
if (myGuess == theNumber)
{
    [C# code block for correct answer]
}
else if (Math.Abs(myGuess - theNumber) <= 1)
{
    [C# code block when burning up - within 1 of correct answer]
}
else if (Math.Abs(myGuess - theNumber) <= 2)
{
    [C# code block when hot - within 2 of correct answer]
}
else if (Math.Abs(myGuess - theNumber) <= 3)
{
    [C# code block when warm - within 3 of correct answer]
}
else
{
    [C# code block when freezing - more than 3 away]
}
```

A last possible change would be to make the project into a math game, and tell the guesser “how far away” the guess is. I’m sure you can think of other ways to change this game. Have fun doing it.



In this class, you learned about a useful input control for numbers, the numeric updown control. You'll learn about other input controls in the next class. And, you learned about a key part of C# programming - decision making. You learned about logical expressions, comparison operators, logical operators, and if structures. And, you will see that the random number object is a fun part of many games. You are well on your way to being a Visual C# programmer.