

# 5

## Labels, Text Boxes, Variables



### Review and Preview

We continue our look at the Visual C# environment and learn some new controls and new C# statements. As you work through this class, remember the three steps for building a Visual C# project: (1) place controls on form, (2) assign properties to controls, and (3) write event methods. In this class, you will examine how to find and eliminate errors in your projects, learn about the label and text box controls, and about C# variables. You will build a project that helps you plan your savings.

## Debugging a Visual C# Project

No matter how well you plan your project and no matter how careful you are in implementing your ideas in the controls and event methods, you will make mistakes. Errors, or what computer programmers call **bugs**, do creep into your project. You, as a programmer, need to have a strategy for finding and eliminating those bugs. The process of eliminating bugs in a project is called **debugging**. Unfortunately, there are not a lot of hard, fast rules for finding bugs in a program. Each programmer has his or her own way of attacking bugs. You will develop your ways. We can come up with some general strategies, though, and that's what we'll give you here.

Project errors, or bugs, can be divided into three types:

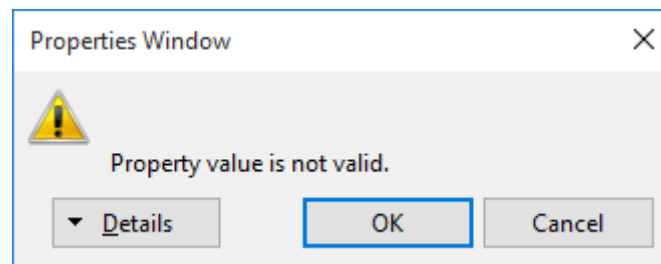
- **Syntax** errors
- **Run-time** errors
- **Logic** errors

**Syntax errors** occur when you make an error setting a property in design mode or when typing a line of C# code. Something is misspelled or something is left out that needs to be there. Your project won't run if there are any syntax errors. **Run-time errors** occur when you try to run your project. It will stop abruptly because something has happened beyond its control. **Logic errors** are the toughest to find. Your project will run OK, but the results it gives are not what you expected. Let's examine each error type and address possible debugging methods.

## Syntax Errors

Syntax errors are the easiest to identify and eliminate. The Visual C# program is a big help in finding syntax errors. Syntax errors will most likely occur as you're setting properties for the controls or writing C# code for event methods.

Start a new project in Visual C#. Go to the project window and try to set the form **Width** property to the word **Junk**. (Click the plus sign next to the **Size** property to see **Width**.) What happened? You should see a little window like this:



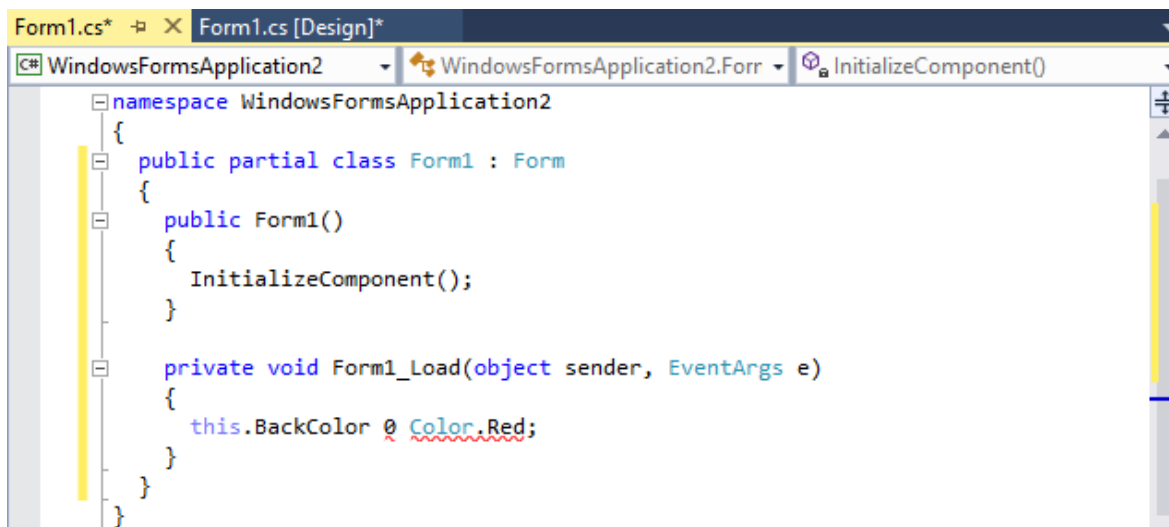
Click **Details** and you will see an explanation of the problem. Remember that property values must be the proper type. Assigning an improper type to a property is a **syntax** error. But, we see Visual C# won't let us make that mistake. Click **Cancel** to restore the **Width** to what it was before you tried to change it.

What happens if you cause a syntax error while writing code. Let's try it.

Establish a **Form1\_Load** event method using the properties window (recall: choose **Events** button in properties window, scroll down to **Load** event and double-click the event name). When the code window opens, under the left curly brace following the header line, type this line, then press <Enter>:

```
this.BackColor 0 Color.Red;
```

This would happen if you typed **0** instead of **=** in the assignment statement. What happened? In the code window, part of the line will appear underlined with a squiggle, similar to what Microsoft Word does when you misspell a word:



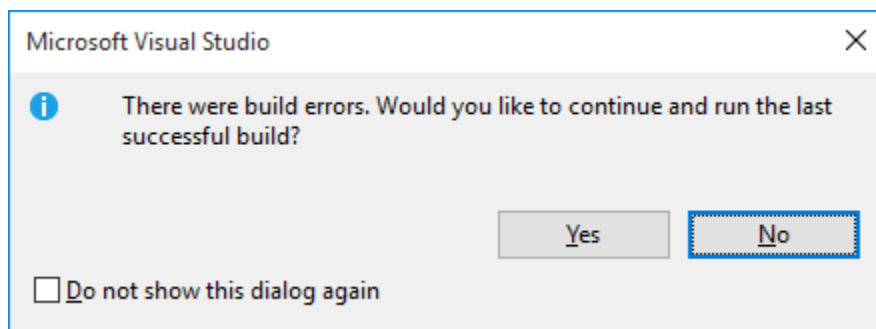
Visual C# has recognized that something is wrong with this statement. You should be able to see what. Any line with an error will be 'squiggled' in places. Placing the cursor over a squiggled line will give some indication of your error.

So, if you make a syntax error, Visual C# will usually know you've done something wrong and make you aware of your mistake. The on-line help system is a good resource for debugging your syntax errors. Note that syntax errors usually result because of incorrect typing - another great reason to improve your typing skills, if they need it.

## Run-Time Errors

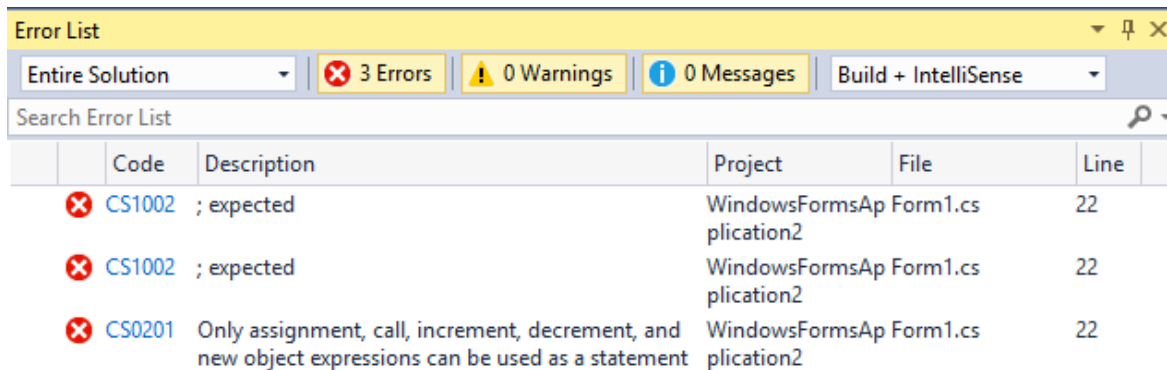
Once you successfully set control properties and write event methods, eliminating all identified syntax errors, you try to run your project. If the project runs, great! But, many times, your project may stop and tell you it found an error - this is a run-time error. You need to figure out why it stopped and fix the problem. Again, Visual C# and on-line help will usually give you enough information to eliminate run-time errors. Let's look at examples.

Working with the same example as above, try to run the project with the incorrect line of code. After you click the **Start** button on the toolbar, the following window should appear:



This tells us an error has occurred in trying to 'build' the project. Click **No** – we don't want to continue. We want to find the error. If 'build errors' occur, they are listed in another Visual C# window – the **Error List**. This list shows you all errors detected in trying to run your program. Go to that window now (if it's not there already, choose **View** in menu, select **Error List**. Error List appears in the Design window. Yours might be floating or docked somewhere.

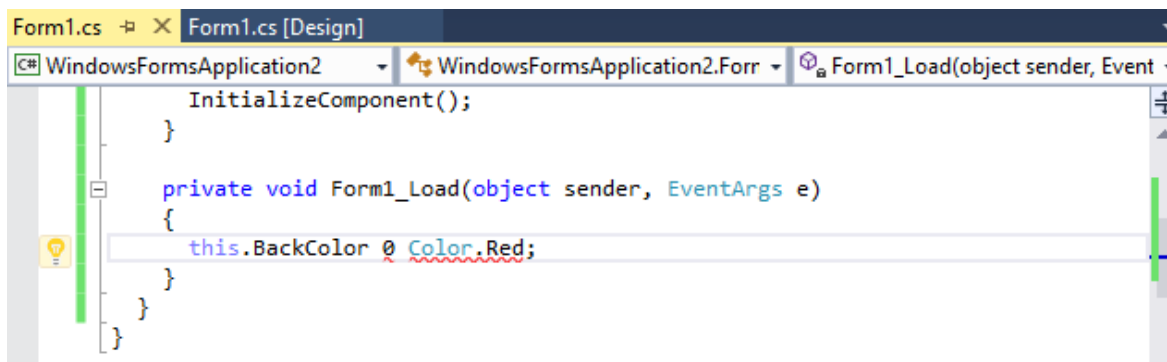
My Error List window is:



The Error List window displays three errors. The first two are CS1002, and the third is CS0201. All errors point to Line 22 of Form1.cs in the WindowsFormsAp plication2 project.

	Code	Description	Project	File	Line
✖	CS1002	; expected	WindowsFormsAp plication2	Form1.cs	22
✖	CS1002	; expected	WindowsFormsAp plication2	Form1.cs	22
✖	CS0201	Only assignment, call, increment, decrement, and new object expressions can be used as a statement	WindowsFormsAp plication2	Form1.cs	22

It has three errors that must be cleared before the program will run. Note each error refers to Line 22 on Form1, pointing to the offending line. Double-click the first error to move to the corresponding line:

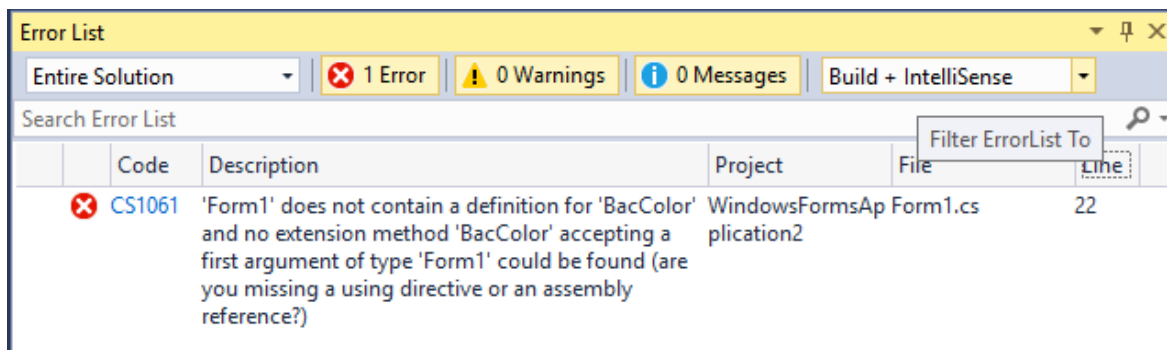


Visual C# is telling you that there is something wrong with how you used this line of code. Using the hints from the task list (the code expected an end-of-line indicator), you should be able to see that the assignment operator (=) is missing. If you don't see the problem, clicking **<F1>** might give you more help.

Let's say we corrected our error by adding the = sign, but we accidentally left out the letter 'k' in the **BackColor** property name, or we typed:

```
this.BacColor = Color.Red;
```

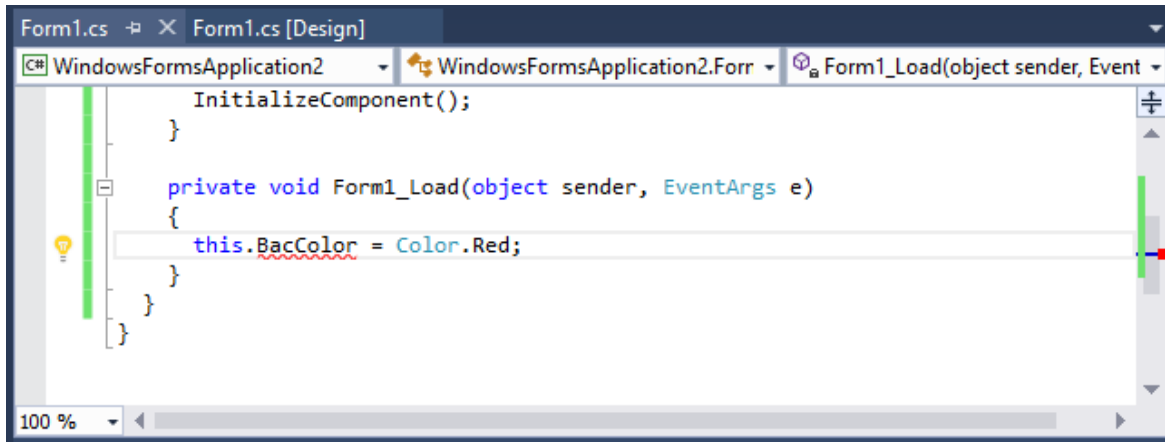
Try running the project and you'll see another 'build error' window. Choose not to continue and go to the Error List:



The message again points to Line 22, saying '**BacColor**' is not defined by the form.



Again, go to Line 22 (double-click the error message) in the code window and you should see:

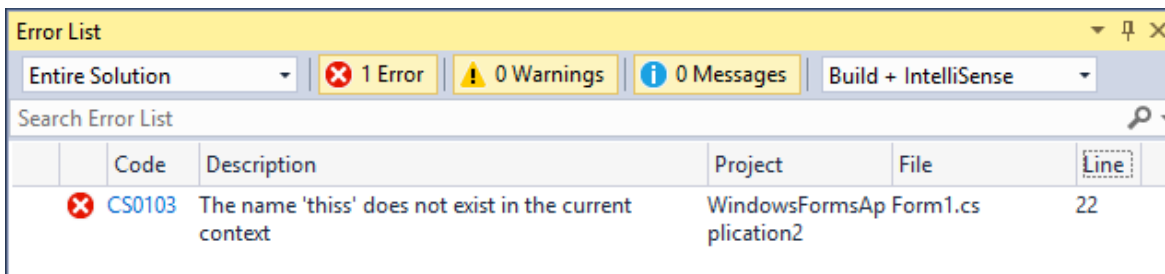


The cursor is next to **Me.BacColor**. Visual C# is telling you it can't find this property for the particular control (the form). You should note the misspelling and correct it.

Now, let's say you correct the property name, but mess up again and type **thiss** instead of **this** when referring to the form:

```
thiss.BackColor = Color.Red;
```

Run the project. You will get another build error, choose not to continue and view the Task list:



The key message here is ‘this does not exist ...’ This usually appears when you have misspelled the assigned name of a control in C# code. Visual C# is trying to assign a property to something using the ‘dot notation’:

```
controlName.PropertyName = Value;
```

But, it can’t find a control with the given name (**this** in this case). Go to the code window, correct the error and run the application. You should finally get a red form!!

The errors we’ve caused here are three of the most common run-time errors: misspelling an assigned Name property, misspelling a property name, or leaving something out of an assignment statement. Notice each run-time error seen was detected prior to running, resulting in a build error. There are other run-time errors that may occur while your application is actually running.

Visual C# refers to some run-time errors as **exceptions**. If a window appears saying you have some kind of exception, the line with the detected error will be shown with suggestions for fixing the error. Be sure to stop the program before trying to fix the error.

We’ve seen a few typical run-time errors. There are others and you’ll see lots of them as you start building projects. But, you’ve seen that Visual C# is pretty helpful in pointing out where errors are and on-line help is always available to explain them. One last thing about run-time errors. Visual C# will not find all errors at once. It will stop at the first run-time error it encounters. After you fix that error, there may be more. You have to fix run-time errors one at a time.

## Logic Errors

Logic errors are the most difficult to find and eliminate. These are errors that don't keep your project from running, but cause incorrect or unexpected results. The only thing you can do at this point, if you suspect logic errors exist, is to dive into your project (primarily, the event methods) and make sure everything is coded exactly as you want it. Finding logic errors is a time-consuming art, not a science. There are no general rules for finding logic errors. Each programmer has his or her own particular way of searching for logic errors.

With the example we have been using, a logic error would be setting the form background color to blue, when you expected red. You would then go into the code to see why this is happening. You would see the color **Color.Blue** instead of the desired value **Color.Red**. Making the change would eliminate the logic error and the form will be red.

Unfortunately, eliminating logic errors is not as easy as this example. But, there is help. Visual C# has something called a **debugger** that helps you in the identification of logic errors. Using the debugger, you can print out properties and other values, stop your code wherever and whenever you want, and run your project line-by-line. Use of the debugger is an advanced topic and will not be talked about in this course. If you want to improve your Visual C# skills, you are encouraged to eventually learn how to use the debugger.

Now, let's improve your skills regarding Visual C# controls. We'll look at two new controls: the **label** and the **text box**.

## Label Control

A **label** is a control that displays information the user cannot edit directly. It is most often used to provide titles for other controls. Or, it is used to display the results of some computer operation. The label control is selected from the toolbox. It appears as:

In Toolbox:



On Form (default properties):



## Properties

A few useful properties for the label are:

<u>Property</u>	<u>Description</u>
<b>Name</b>	Name used to identify label. Three letter prefix for label names is <b>lbl</b> .
<b>Text</b>	Text (string type) that appears in the label.
<b>TextAlign</b>	Specifies how the label text is positioned.
<b>Font</b>	Sets style, size, and type of Text text.
<b>BackColor</b>	Sets label background color.
<b>ForeColor</b>	Sets color of Text text.
<b>Left</b>	Distance from left side of form to left side of label (referred to by X in properties window, expand <b>Location</b> property).
<b>Top</b>	Distance from top side of form to top side of label (referred to by Y in properties window, expand <b>Location</b> property).

---

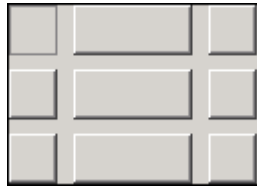
<b>Width</b>	Width of the label in pixels (expand <b>Size</b> property).
<b>Height</b>	Height of label in pixels (expand <b>Size</b> property).
<b>BorderStyle</b>	Determines type of label border.
<b>Visible</b>	Determines whether the label appears on the form (in run mode).
<b>AutoSize</b>	If <b>True</b> (default value), label adjusts to size of text. If <b>False</b> , label can be resized.

Note, by default, the label control has no resizing handles. To resize the label, set `AutoSize` to `False`.

## Example

Make sure Visual C# is running and start a new project. Put a label on the form. Resize it (`AutoSize` must be `False` to do this) and move it, if desired. Set the `Text` property. Try different `Fonts`. See the difference among the `BorderStyle` possibilities; notice the default value (**None**) makes the button match with the form, **Fixed Single** places a box around the label, and **Fixed3D** gives the label a three-dimensional inset look. Change the `BackColor` and `ForeColor` properties. You may find certain color combinations that don't do a very good job of displaying the `Text` when in color. Make sure you are aware of combinations that do and don't work. You want your user to be able to read what is displayed.

The most used label property is `Text`. It holds the information that is displayed in the label control. There are two things you need to be aware of. First, by default, the label will 'grow' to hold any `Text` you might provide for it. If the label size is not acceptable, you can try things like changing **Font** or **AutoSize**. If the label is made to be larger than the text it holds (by setting `AutoSize` to `False`), you will also want to set the `TextAlign` property. Try different values of the `TextAlign` property; there are nine different alignments selected from a 'graphical' menu:



Vertical choices are: top, middle, and bottom justification. Horizontal choices are: left, center, and right justification.

The second thing you need to know is that **Text** is a string type property. It can only hold string values. When setting the **Text** property in run mode, the **Text** information must be in quotes. For example, if you have a label control named **lblExample** and you want to set the **Text** property to **My Label Box**, you would use the C# code (note the dot notation):

```
lblExample.Text = "My Label Box";
```

You don't have to worry about the quotes when setting the **Text** in design mode. Visual C# knows this is a string value.

## Events

There is only one label event of interest:

<u>Event</u>	<u>Description</u>
<b>Click</b>	Event executed when user clicks on the label with the mouse.

With this event, you could allow your user to choose among a set of displayed label boxes. Why would you want to do this? Example applications include multiple choice answers in a test or color choices.

## Typical Use of Label Control

The usual design steps for the label control to display unchanging text (for example, to provide titling information) are:

- Set the **Name** (though not really necessary since you rarely write code for a label control) and **Text** property.
- You may also want to change the **Font**, **BackColor** and **ForeColor** properties.

To use the label control for changing text, for example, to show some computed results, use these steps:

- Set the **Name** property. Initialize **Text** to desired string.
- Set **AutoSize** to **False**, resize control and select desired value for **TextAlign**.
- Assign **Text** property (string type) in code where needed.
- You may also want to change the **Font**, **BackColor** and **ForeColor** properties.



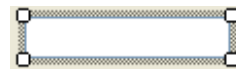
## Text Box Control

The **text box** control is used to display information entered in design mode, by a user in run mode, or assigned within an event method. Just think of a text box as a label whose contents your user can (or may not be able to) change. The text box is selected from the Visual C# toolbox. It appears as:

In Toolbox:



On Form (default properties):



## Properties

The text box has a wealth of useful properties:

<u>Property</u>	<u>Description</u>
<b>Name</b>	Name used to identify text box. Three letter prefix for text box names is <b>txt</b> .
<b>Text</b>	Text (string value) that appears in text box.
<b>TextAlign</b>	Sets whether Text is left-justified, right-justified, or centered in text box.
<b>Font</b>	Sets style, size, and type of Text.
<b>MultiLine</b>	Specifies whether text box displays one line or multiple lines.
<b>ScrollBars</b>	Specifies type of displayed scroll bar(s).
<b>MaxLength</b>	Maximum length of displayed Text. If <b>0</b> , length is unlimited.
<b>BackColor</b>	Sets text box background color.
<b>ForeColor</b>	Sets color of Text.

<b>Left</b>	Distance from left side of form to left side of text box (X in the properties window, expand <b>Location</b> property).
<b>Top</b>	Distance from top side of form to top side of text box (Y in the properties window, expand <b>Location</b> property).
<b>Width</b>	Width of the text box in pixels (expand <b>Size</b> property).
<b>Height</b>	Height of text box in pixels (expand <b>Size</b> property).
<b>ReadOnly</b>	If <b>True</b> , user can't change contents of text box (run mode only).
<b>TabStop</b>	If <b>False</b> , the control cannot be 'tabbed' to.
<b>BorderStyle</b>	Determines type of text box border.
<b>Visible</b>	Determines whether the text box appears on the form (in run mode).

## Example

Start a new Visual C# project. Put a text box on the form. Resize it and move it, if desired. Set the Text property. Try different Fonts. Try different values of the TextAlign property. See the difference among the BorderStyle possibilities. The label box used **None** as default, the text box uses **Fixed3D**. Change the BackColor and ForeColor properties. Set MultiLine to **True** and try different ScrollBars values. I think you can see the text box is very flexible in how it appears on your form.

Like the Text property of the label control, the Text property of a text box is a string value. So, when setting the Text property in run mode, we must enclose the value in quotes (") to provide a proper assignment. Setting the Text property in design mode does not require (and you shouldn't use) quotes.

## Events

The most important property of the text box is the **Text** property. As a programmer, you need to know when this property has changed in order to make use of the new value. There are two events you can use to do this:

<u>Event</u>	<u>Description</u>
<b>TextChanged</b>	Event executed whenever <b>Text</b> changes.
<b>Leave</b>	Event executed when the user leaves the text box and causes an event on another control.

The **TextChanged** event is executed a lot - every time a user presses a key while typing in the text box, the **TextChanged** event method is called. Looking at the **Text** property in this event method will give you its current value.

The **Leave** event is the more useful event for examining **Text**. Remember in placing controls on the form in design mode, you can make one control 'active' by clicking on it. There is a similar concept while an application is in run mode. A user can have interaction with only one control at a time. The control the user is interacting with (causing events) is said to have **focus**. While a user is typing in a text box, that box has focus. The **Leave** event is executed when you leave the text box and another control gets focus. At that point, we know the user is done typing in the text box and is done changing the **Text** property. That's why this event method is a good place to find the value of the **Text** property.

## Typical Use of Text Box Control

There are two primary ways to use a text box – as an input control or as a display control. If the text box is used to accept some input from the user, the usual design steps:

- Set the **Name** property. Initialize **Text** property to desired string.
- If it is possible to input multiple lines, set **MultiLine** property to **True**. Also, set **ScrollBars** property, if desired.
- You may also want to change the **Font**, **BackColor** and **ForeColor** properties.

If using the control just to display some information (no user modification possible), follow these usual design steps:

- Set the **Name** property. Initialize **Text** property to desired string.
- Set **ReadOnly** property to **True** (once you do this, note the background color will change).
- Set **TabStop** to False.
- If displaying more than one line, set **MultiLine** property to **True**.
- Assign **Text** property in code where needed.
- You may also want to change the **Font**, **BackColor** and **ForeColor** properties.

## C# - The Second Lesson

In this class, you will learn some new C# concepts. We will discuss variables (name, type, declaring), arithmetic operations, and some functions and techniques for working with strings.

### Variables

All computer programs work with information of one kind or another. Numbers, text, colors and pictures are typical types of information they work with. Computer programs need places to store this information while working with it. We have seen one type of storage used by Visual C# projects - control properties. Control properties store information like control size, control appearance, control position on the form, and control colors.

But, control properties are not sufficient to store all information a project might need. What if we need to know how much ten bananas cost if they are 25 cents each? We would need a place to store the number of bananas, the cost of each banana, and the result of multiplying these two numbers together. To store information other than control properties in Visual C# projects, we use something called **variables**. They are called variables because the information stored there can change, or vary, during program execution. Variables are the primary method for moving information around in a Visual C# project. And, certain rules must be followed in the use of variables. These rules are very similar to those we have already established for control properties.

## Variable Names

You must **name** every variable you use in your project. Rules for naming variables are:

- No more than 40 characters.
- Can only use letters, numbers, and the underscore (\_) character.
- The first character must be a letter. It is customary, though not required, in Visual C# that this first letter be lower case.
- You cannot use a word reserved by Visual C# (for example, you can't have a variable named Form or one named Beep).

The most important rule is to use variable names that are meaningful. You should be able to identify the information stored in a variable by looking at its name. As an example, in our banana buying example, good names would be:

<u>Quantity</u>	<u>Variable Name</u>
Cost of each banana	bananaCost
Number of bananas purchased	bananas
Cost of all bananas	totalBananaCost

As mentioned in an earlier class, the Visual C# language is case sensitive. This means the names **BananaCost** and **bananacost** refer to different variables. Make sure you assign unique, easily identified, names to each variable. As with control names, we suggest mixing upper and lower case letters for improved readability. You will notice, as you type code, that the Visual C# editor will adjust the case of control names, variables and reserved C# keywords, as necessary.

## Variable Types

We need to know the **type** of information stored by each variable. The same types used for properties can be applied to variables: **int** (integer), **bool** (Boolean) and **string**. There are other types too - consult on-line help for types you might want to use.

Here, we look at one more type we will use with variables: the **double** type. Up to now, all the projects we've worked with have used integer (or whole number) values. But, we know most 'real-world' mathematics involves decimal numbers. The double type is just that - a number that has a decimal point. In computer language, we call it a **floating point number**. The 'point' that is floating (moving around) is the decimal. Examples of double type numbers are:

2.00            -1.2            3.14159

Variables can appear in assignment statements:

```
variableName = NewValue;
```

Only a single variable can be on the left side of the assignment operator (=) while any legal C# expression, using any number of variables, can be on the right side of the operator. Recall that, in this statement, **NewValue** is evaluated first, then assigned to **variableName**. The major thing we need to be concerned with is that NewValue is the same **type** as variableName. That is, we must assign a properly typed value to the variable. This is the same thing we had to do with property values.



## Declaring Variables

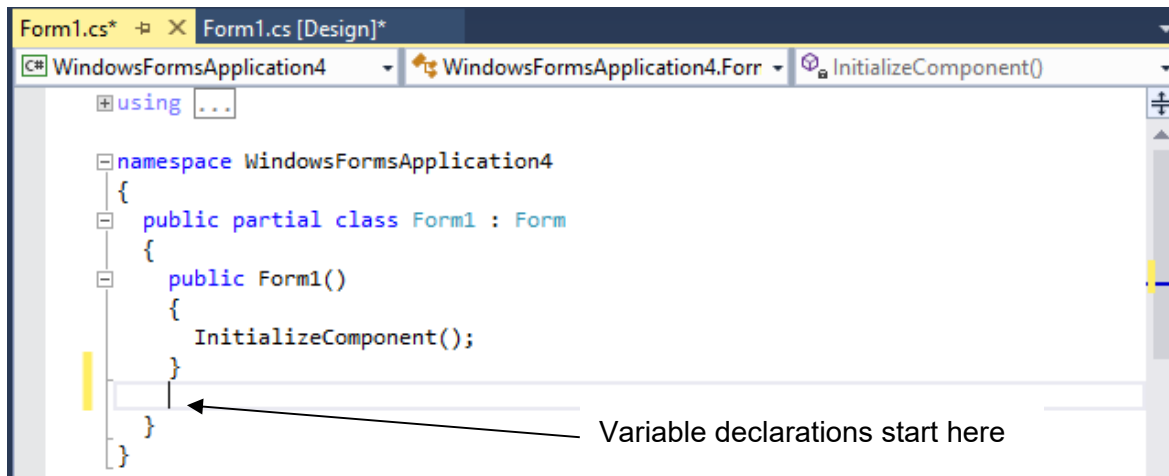


Once we have named a variable and determined what type we want it to be, we must relay this information to our Visual C# project. We need to **declare** our variables. (We don't have to declare control properties since Visual C# already knows about them.) The statement used to declare a variable named **variableName** as type **type** is:

```
type variableName;
```

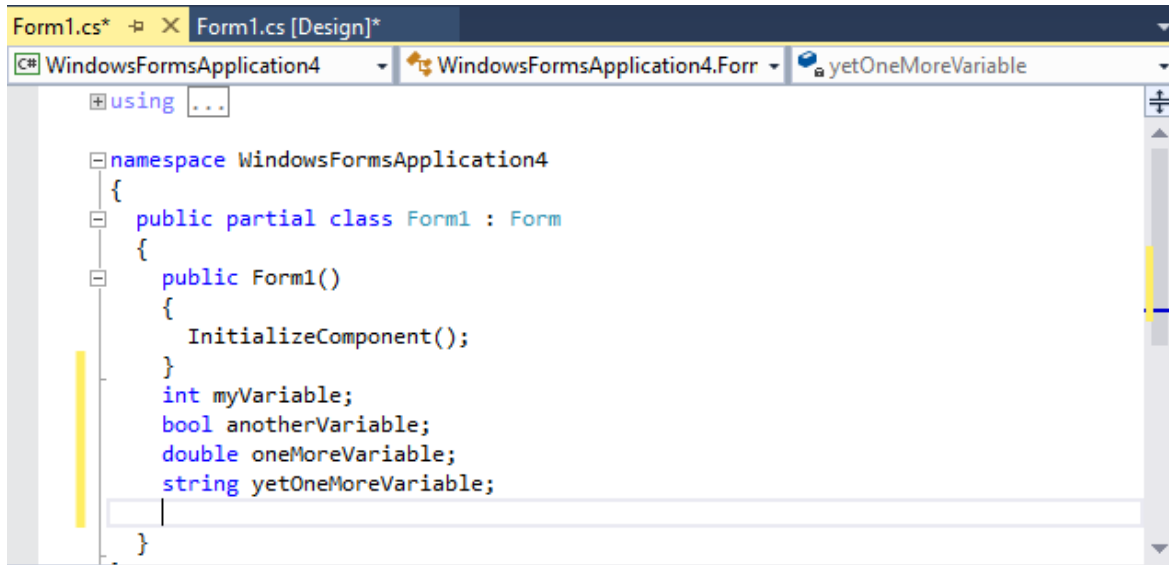
We need a declaration statement like this for every variable in our project. This may seem like a lot of work, but it is worth it. Proper variable declaration makes programming easier, minimizes the possibility of program errors, and makes later program modification easier.

So, where do we put these variable declarations. Start a new Visual C# project and bring up the code window. The code window will look like this:



We will put variable declaration statements directly beneath form constructor code (**Public Form1**) and before any event methods. This location in the code window is known as the **general declarations** area and any variables declared here can be used (the value can be accessed and/or changed) in any of the project's event methods.

Try typing some variable declarations in the code window. Here are some examples to try:



```
Form1.cs*  Form1.cs [Design]*
C# WindowsFormsApplication4  WindowsFormsApplication4.Forr  yetOneMoreVariable
using ...

namespace WindowsFormsApplication4
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        int myVariable;
        bool anotherVariable;
        double oneMoreVariable;
        string yetOneMoreVariable;
    }
}
```

## Type Casting

In each assignment statement, it is important that the type of data on both sides of the operator (=) is the same. That is, if the variable on the left side of the operator is an int, the result of the expression on the right side should be int. Visual C# (by default) will try to do any conversions for you. When it can't, an error message will be printed. In those cases, you need to explicitly **cast** the result. This means convert the right side to the same side as the left side. Assuming the desired type is **type**, the casting statement is:

```
leftSide = (type) rightSide;
```

You can cast from any basic type (decimal and integer numbers) to any other basic type. For example, in the statement:

```
a = (int) (3.14159);
```

The variable **a** (an **int** type) will be assigned a value of 3.

Be careful when casting from higher precision numbers to lower precision numbers. Problems arise when you are outside the range of numbers.

## Arithmetic Operators

One thing computer programs are very good at is doing arithmetic. They can add, subtract, multiply, and divide numbers very quickly. We need to know how to make our Visual C# projects do arithmetic. There are five **arithmetic operators** in the C# language.

**Addition** is done using the plus (+) sign and **subtraction** is done using the minus (-) sign. Simple examples are:

Operation	Example	Result
Addition	$7 + 2$	9
Addition	$3 + 8$	11
Subtraction	$6 - 4$	2
Subtraction	$11 - 7$	4

**Multiplication** is done using the asterisk (\*) and **division** is done using the slash (/). Simple examples are:

Operation	Example	Result
Multiplication	$8 * 4$	32
Multiplication	$2 * 12$	24
Division	$12 / 2$	6
Division	$42 / 6$	7

I'm sure you've done addition, subtraction, multiplication, and division before and understand how each operation works. The other arithmetic operator may not be familiar to you, though.

The other arithmetic operator we use is called the remainder operator (%). This operator gives you the remainder that results from dividing two whole numbers. It may not be obvious now, but the remainder operator is used a lot in computer programming. Examples:

Example	Division Result	Remainder Result
7 % 4	1 Remainder 3	3
14 % 3	4 Remainder 2	2
25 % 5	5 Remainder 0	0

Study these examples so you understand how the remainder operator works in C#.

What happens if an assignment statement contains more than one arithmetic operator? Does it make any difference? Look at this example:

$$7 + 3 * 4$$

What's the answer? Well, it depends. If you work left to right and add 7 and 3 first, then multiply by 4, the answer is 40. If you multiply 3 times 4 first, then add 7, the answer is 19. Confusing? Well, yes. But, C# takes away the possibility of such confusion by having rules of **precedence**. This means there is a specific order in which arithmetic operations will be performed. That order is:

1. Multiplication (\*) and division (/)
2. Remainder (%)
3. Addition (+) and subtraction (-)

So, in an assignment statement, all multiplications and divisions are done first, then remainder operations, and lastly, additions and subtractions. In our example

(7 + 3 \* 4), we see the multiplication will be done before the addition, so the answer provided by C# would be 19.

If two operators have the same precedence level, for example, multiplication and division, the operations are done left to right in the assignment statement. For example:

$$24 / 2 * 3$$

The division (24 / 2) is done first yielding a 12, then the multiplication (12 \* 3), so the answer is 36. But what if we want to do the multiplication before the division - can that be done? Yes - using the C# **grouping operators** - parentheses (). By using parentheses in an assignment statement, you force operations within the parentheses to be done first. So, if we rewrite our example as:

$$24 / (2 * 3)$$

the multiplication (2 \* 3) will be done first yielding 6, then the division (24 / 6), yielding the desired result of 4. You can use as many parentheses as you want, but make sure they are always in pairs - every left parenthesis needs a right parenthesis. If you nest parentheses, that is have one set inside another, evaluation will start with the innermost set of parentheses and move outward. For example, look at:

$$((2 + 4) * 6) + 7$$

The addition of 2 and 4 is done first, yielding a 6, which is multiplied by 6, yielding 36. This result is then added to 7, with the final answer being 43. You might also want to use parentheses even if they don't change precedence. Many times, they are used just to clarify what is going on in an assignment statement.

As you improve your programming skills, make sure you know how each of the arithmetic operators work, what the precedence order is, and how to use parentheses. Always double-check your assignment statements to make sure they are providing the results you want.

Some examples of C# assignment statements with arithmetic operators:

```
totalBananaCost = numberBananas * bananaCost;  
numberOfWeeks = numberOfDays / 7;  
averageScore = (score1 + score2 + score3) / 3.0;
```

Notice a couple of things here. First, notice the parentheses in the **averageScore** calculation forces C# to add the three scores before dividing by 3. Also, notice the use of “white space,” spaces separating operators from variables. This is a common practice in C# that helps code be more readable. We’ll see lots and lots of examples of assignment statements as we build projects in this course.



## String/Number Conversion Methods

A common task in any Visual C# project is to take numbers input by the user, do some arithmetic operations on those numbers, and output the results of those operations. How do you do this? With the Visual C# knowledge you have up to this point, you probably see you could use text box controls to allow the user to input numbers. Then you could use the arithmetic operators to do the math and label controls to display the results of the math. And, that's just what you would do. But, there are two problems:

**Problem One:** Arithmetic operators can only work with numbers (for example, integer variables and integer properties), but the value provided by a text box control (the Text property) is a string. You can't add and multiply string type variables and properties!

**Problem Two:** The result of arithmetic operations is a number. But the Text property of a label control (where we want to display these results) is a string type. You can't store numerical data in a string quantity!

We need solutions to these two problems. The solutions lie in the **C# built-in methods**. We need ways to convert strings to numbers and, conversely, numbers to strings. With this ability, we could take the Text property from a text box, convert it to a number, do some math, and convert that numerical result to a string that could be used as a Text property in a label box. This is a very common task in C# and C# has a large set of methods that help us do such common tasks. We will look at these in a bit, but first let's define just what a method is.

A C# method is a built-in procedure that, given some information by us, computes some desired value. The format for using a method is:

```
methodValue = MethodName(ArgumentList);
```

**MethodName** is the name of the method and **ArgumentList** is a list of values (separated by commas) provided to the function so it can do its work. In this assignment statement, **MethodName** uses the values in **ArgumentList** to compute a result and assign that result to the variable we have named **methodValue**. We must insure the variable **methodValue** has the same type as the value computed by **MethodName**. How do we know what C# functions exist, what type of information they provide, and what type of **arguments** they require? Use the Visual C# on-line help system and search for **Methods**. You'll see that there are lots of them. We'll cover some of them in this class, but you'll have to do a little studying on your own to learn about most of them. Now, let's look at some C# methods that help in converting numbers to strings and vice versa.

There are two C# methods we will use to convert a string type variable (or control property) to a numerical value. The method **Convert.ToInt32** method converts a **string** type to an **int** type (the 32 implies 32 bits are used to store an int type). The format for using this function is:

```
yourNumber = Convert.ToInt32(yourString);
```

The **Convert.ToInt32** method takes the **yourString** variable (remember this is called an argument of the method), converts it to a numerical value, and assigns it to the variable **yourNumber** (which must be an int type). We could then use **yourNumber** in any arithmetic statement. Recall strings must be enclosed in quotes. An example using this method:

```
yourNumber = Convert.ToInt32("23");
```

Following this assignment statement, the variable `yourNumber` has a numerical value of 23.

The corresponding method that converts a **string** to a **double** type is `Convert.ToDouble`. The format for using this function is:

```
yourNumber = Convert.ToDouble(yourString);
```

The **Convert.ToDouble** method takes the **yourString** variable, converts it to a numerical value, and assigns it to the variable **yourNumber** (which must be a double type). We could then use `yourNumber` in any arithmetic statement. An example using this method:

```
yourNumber = Convert.ToDouble("3.14159");
```

Following this assignment statement, the variable `yourNumber` has a numerical value of 3.14159.

The C# **Convert.ToString** method will convert any numerical variable (or control property) to a **string**. The format for using this method is:

```
yourString = Convert.ToString(yourNumber);
```

The **Convert.ToString** method takes the **yourNumber** argument, converts it to a string type value, and assigns it to the string variable named **yourString**. In the example:

```
yourString = Convert.ToString(23);
```

the variable `yourString` has a string value of "23". And, with:

```
yourString = Convert.ToString(3.14159);
```

the variable `yourString` has a string value of "3.14159".

You should be comfortable with converting numbers to strings and strings to numbers using these methods. As mentioned, this is one of the more common tasks you will use when developing Visual C# projects.

## String Concatenation

A confession - in the above discussion, you were told a little lie. The statement was made that you couldn't add and multiply strings. Well, you can't multiply them, but you can do something similar to addition. Many times in Visual C# projects, you want to take a string variable from one place and 'tack it on the end' of another string. The fancy word for this is **string concatenation**. . The concatenation operator is a plus sign (+) and it is easy to use. As an example:

```
newString = "Visual C# " + "is Fun!";
```

After this statement, the string variable **newString** will have the value "Visual C# is Fun!".

Notice the string concatenation operator is identical to the addition operator. We always need to insure there is no confusion when using both. As you've seen, string variables are a big part of Visual C#. As you develop as a programmer, you need to become comfortable with strings and working with them. You're now ready to attack a new project.

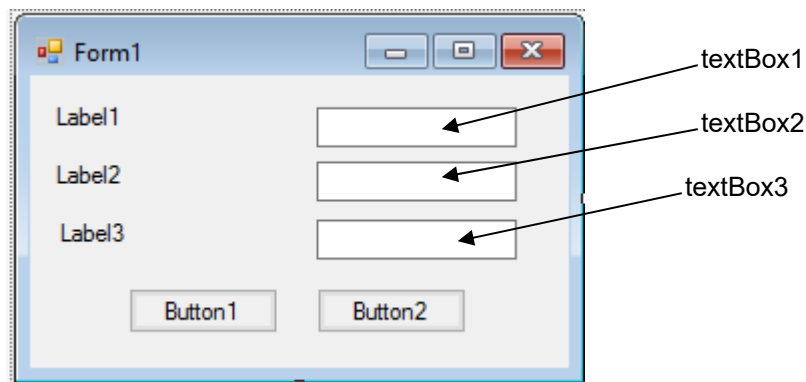
## Project - Savings Account

### Project Design

In this project, we will build a savings account calculator. We will input how much money we can put into an account each week and the number of weeks we put money in the account. The project will then compute how much we saved. We will use text boxes as both the input controls and for output information. A button will be used to do the computation. This project is saved as **Savings** in the course projects folder (**\BeginVCS\BVCS Projects**).

### Place Controls on Form

Start a new project in Visual C#. Place three text box controls, three label controls, and two buttons on the form. Your form should resemble this:



Again, try using copy and paste for the similar controls.

## Set Control Properties

Set the control properties using the properties window (remember, controls are listed by their default name):

**Form1** Form:

Property Name	Property Value
Text	Savings Account
FormBorderStyle	Fixed Single
StartPosition	CenterScreen

**label1** Label:

Property Name	Property Value
Name	lblDepositHeading
Text	Weekly Deposit
Font	Arial
Font Size	10

**label2** Label:

Property Name	Property Value
Name	lblWeeksHeading
Text	Number of Weeks
Font	Arial
Font Size	10

**label3** Label:

Property Name	Property Value
Name	lblTotalHeading
Text	Total Savings
Font	Arial
Font Size	10

**textbox1** Text Box:

Property Name	Property Value
Name	txtDeposit
TextAlign	Right
Font	Arial
Font Size	10

**textbox2** Text Box:

Property Name	Property Value
Name	txtWeeks
TextAlign	Right
Font	Arial
Font Size	10

**textbox3** Text Box:

Property Name	Property Value
Name	txtTotal
TextAlign	Right
Font	Arial
Font Size	10
ReadOnly	True
BackColor	White
TabStop	False

(Note the background color changes when setting ReadOnly to True. Hence, we set BackColor to White to match the appearance of the other two text boxes.)

**button1** Button:

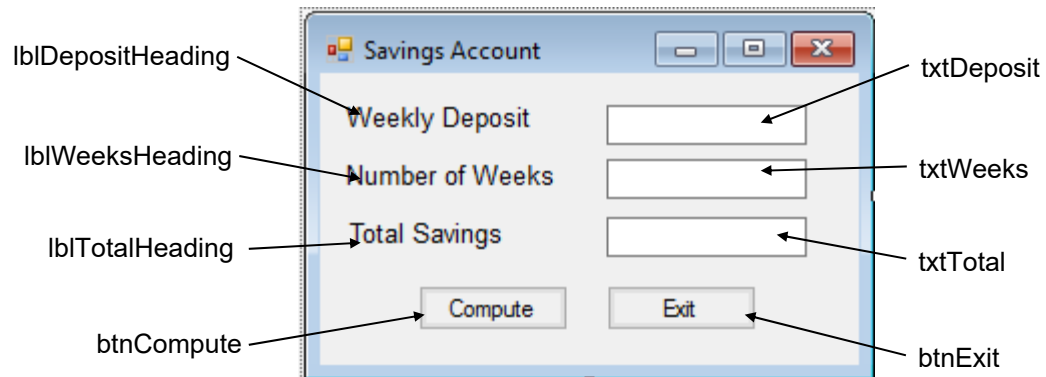
Property Name	Property Value
Name	btnCompute
Text	Compute

**button2** Button:

Property Name	Property Value
Name	btnExit
Text	Exit



Note this is the first time you have been asked to change Font properties. Review the procedure for doing this (Class 4 under Button Control), if necessary. Change any other properties, like colors, if you would like. When you are done, your form should resemble this:



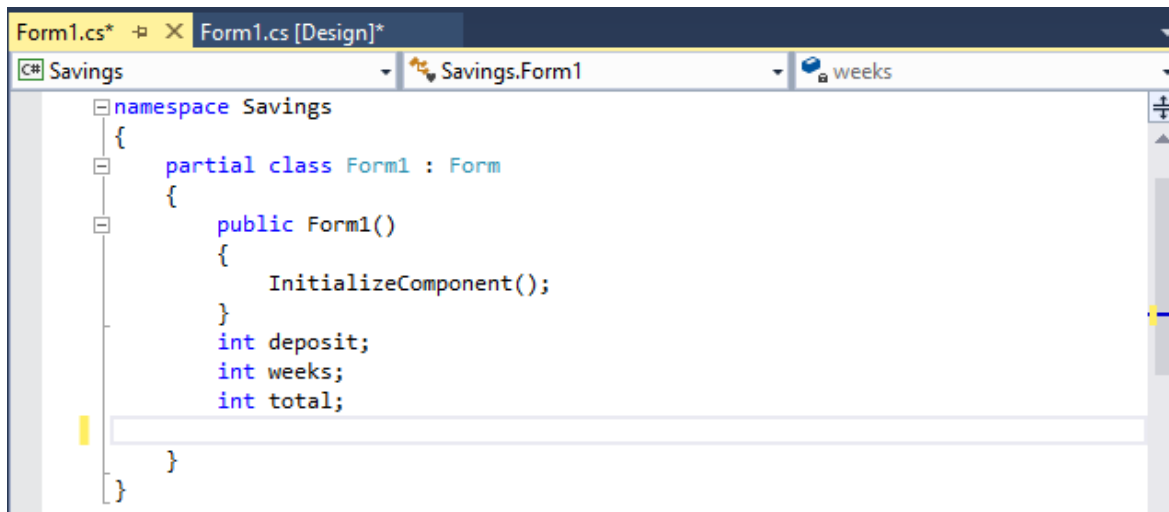
## Write Event Procedures

In this project, the user types an amount in the **Weekly Deposit** text box. Then, the user types a value in the **Number of Weeks** text box. Following this, the user clicks the **Compute** button. The project determines the total amount in the savings account and displays it in the lower text box control. Hence, the primary event in this project is the **Click** event on the Compute button. The only other event is the **Click** event on the **Exit** button. It's always good to have an obvious way for the user to exit a project.

We need three variables in this project (we will use **int** types), one to hold the weekly deposit amount (**deposit**), one to store the number of weeks (**weeks**), and one to store the total savings (**total**). Open the code window and find the **general declarations** area (the area right under the form constructor code). Declare these three variables:

```
int deposit;  
int weeks;  
int total;
```

The code window should appear as:



The event methods start after the variable declarations.

The **btnCompute\_Click** event implements the following steps:

1. Convert input deposit value (**txtDeposit.Text**) to a number and store it in the variable **deposit**.
2. Convert input number of weeks (**txtWeeks.Text**) to a number and store it in the variable **weeks**.
3. Multiply deposit times weeks and store the result in the variable **total**.
4. Convert the numerical value **total** to a string, concatenate it with a dollar sign (\$), and store it in **Text** property of **txtTotal**.

Establish the **btnCompute\_Click** event method using the properties window and type this code (which translates the steps above):

```
private void btnCompute_Click(object sender, EventArgs e)
{
    // Get deposit amount
    deposit = Convert.ToInt32(txtDeposit.Text);
    // Get number of weeks
    weeks = Convert.ToInt32(txtWeeks.Text);
    // Compute total savings
    total = deposit * weeks;
    // Display Total
    txtTotal.Text = "$" + Convert.ToString(total);
}
```

Notice how is easy it is to translate the listed steps to actual C# code. It is just paying attention to details. In particular, look at the use of `Convert.ToInt32` and `Convert.ToString` for string-number conversion.

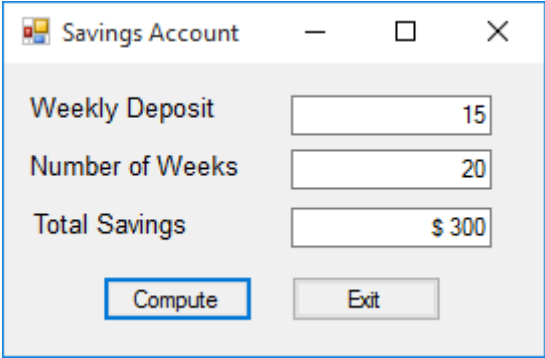
The **btnExit\_Click** event method is simply one line of code that stops the program by closing the form:

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

Save your project by clicking the **Save All** button.

## Run the Project

Run the project. Click in the **Weekly Deposit** text box and type some value. Do the same with **Number of Weeks**. Click the **Compute** button. Your answer should appear in the **Total** text box control. Make sure the answer is correct. Remember, a big step in project design is making sure your project works correctly! If you say you want to save 10 dollars a week for 10 weeks and your computer project says you will have a million dollars by that time, you should know something is wrong somewhere! Click **Exit** to make sure it works. Save your project if you changed anything. Here's a run I made:



Label	Value
Weekly Deposit	15
Number of Weeks	20
Total Savings	\$ 300

Buttons: Compute, Exit

This project may not seem all that complicated. And it isn't. After all, we only multiplied two numbers together. But, the project demonstrates steps that are used in every Visual C# project. Valuable experience has been gained in recognizing how to read input values, convert them to the proper type, do the math to obtain desired results, and output those results to the user.

## Other Things to Try

Most savings accounts yield interest, that is the bank actually pays you for letting them use your money. This savings account project has ignored interest. But, it is fairly easy to make the needed modifications to account for interest - the math is just a little more complicated. We will give you the steps, but not show you how, to change your project. Give it a try if you'd like:

- Define a variable **interest** to store the yearly savings interest rate. Interest rates are decimal numbers, so use the **double** type for this variable (it's the first time we've used decimals!).
- Add another text box to allow the user to input this interest rate. Name it **txtInterest**.
- Add a label control to identify the new text box (set the **Text** to **Interest Rate**).
- Modify the code to use interest in computing **total**. interest is found using:

```
interest = Convert.ToDouble(txtInterest.Text);
```

Then, **total** (get ready - it's messy looking) is computed using:

```
total = (int) (5200 * (deposit * (Math.Pow((1 + interest  
/ 5200), weeks) - 1) / interest));
```

Make sure you type this all on one line - the word processor has made it look like it is on two. As we said, this is a pretty messy expression, but it's good practice in using parentheses and arithmetic operators. Note also the use of the **int** keyword to convert (cast) the computed result to an integer number. The number '5200' is used here to convert the interest from a yearly value to a weekly value.

This equation uses a C# method that we haven't seen yet, the **Pow** method, also called the **exponentiation** method. In exponentiation, a number is multiplied times itself a certain number of times. If we multiply a number by itself 4 times, we say we raise that number to the 4<sup>th</sup> power. The C# method used for exponentiation is:

```
Math.Pow(argument1, argument2)
```

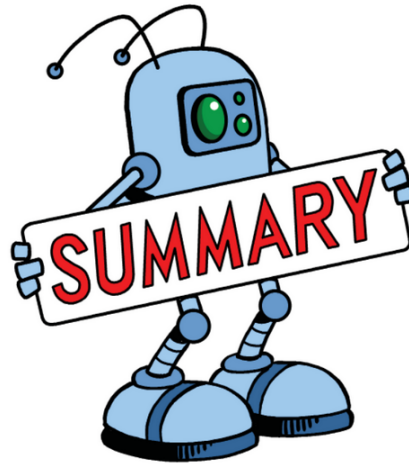
Notice the **Pow** (stands for power) function has two arguments. **argument1** is the number we are multiplying times itself **argument2** times. In other words, this method raises argument1 to the argument2 power. Each argument and the returned value are **double** type numbers. Some examples:

<b>Example</b>	<b>Result</b>
<b>Math.Pow(4.0, 2.0)</b>	16.0
<b>Math.Pow(-3.0, 3.0)</b>	-27.0
<b>Math.Pow(10.0, 4.0)</b>	10000.0

In each example here, the arguments have no decimal parts. We have done this to make the examples clear. You are not limited to such values. It is possible to use this function to compute what happens if you multiply 7.654 times itself 3.16 times!! (The answer is 620.99, by the way.)

Now, run the modified project. Type in values for deposit, weeks, and interest. Click the Compute button. Make sure you get reasonable answers. (As a check, if you use a Deposit value of 10, a Weeks value of 20, and an Interest value of 6.5, the Total answer should be \$202 - note you'd have \$200 without interest, so this makes sense). The project converts total to an integer (using the cast) even though there is probably a decimal (some cents) involved in the answer. Save your project.





In this class, you have learned a lot of new material. You learned about the label and text box controls. You learned about variables: naming them, their types and how to declare them properly. And, you learned functions that allow you to change from string variables to numbers and from number to strings. You learned how to do arithmetic in C#. Like we said, a lot of new material. In subsequent classes, we will stress new controls and new C# statements more than new features about the Visual C# environment. You should be fairly comfortable in that environment, by now.