Debugging your Applications

Debugging refers to the process of trying to track down errors in your programmes. It can also refer to handling potential errors that may occur. There are three types of errors that we'll take a look at:

- Design-time errors
- Run-Time errors
- Logical errors

The longer your code gets, the harder it is to track down why things are not working. By the end of this section, you should have a good idea of where to start looking for problems. But bear in mind that debugging can be an art in itself, and it gets easier with practice.

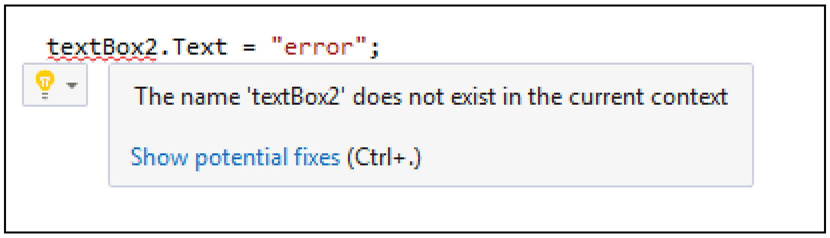
Errors at Design-Time

Design-Time errors are ones that you make before the programme even runs. In fact, for Design-Time errors, the programme won't run at all, most of the time. You'll get a popup message telling you that there were build errors, and asking would you like to continue.

Design-Time errors are easy enough to spot because the C# software will underline them with a wavy coloured line. You'll see three different coloured lines: blue, red and green. The blue wavy lines are known as **Edit and Continue** issues, meaning that you can make change to your code without having to stop the programme altogether. Red wavy lines are **Syntax** errors, such as a missing semicolon at the end of a line, or a missing curly bracket in an IF Statement. Green wavy lines are **Compiler Warnings**. You get these when C# spots something that could potentially cause a problem, such as declaring a variable that's never used.

Red Wavy Lines

In the image below, you can see that there's a red wavy line under textBox2 (earlier versions of Visual Studio may have blue wavy lines, instead of red ones):



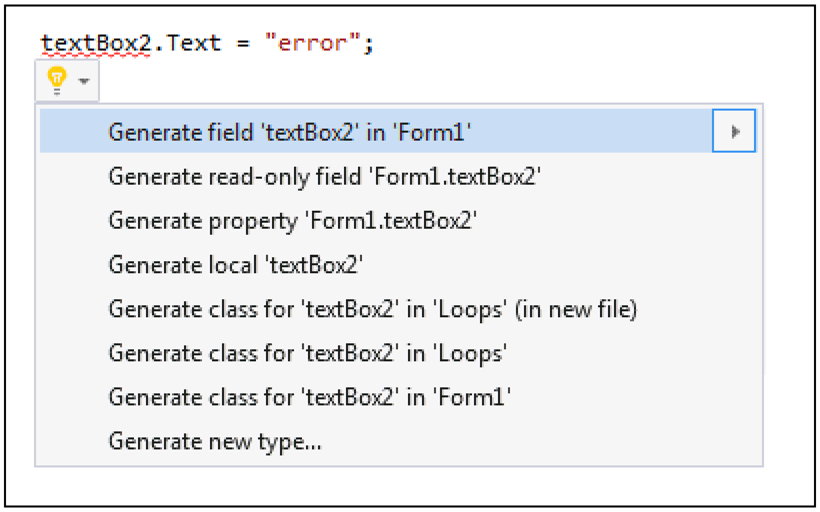
```
textBox2.Text = "error";
```

The name 'textBox2' does not exist in the current context

Show potential fixes (Ctrl+.)

This is an Edit and Continue error. It's been flagged because the form doesn't have a control called textBox2 - it's called textBox1. We can simply delete the 2 and replace it with a 1. The programme can then run successfully. Holding your mouse over the wavy underline gives an explanation of the error.

Click the arrow next to the light bulb to see a list of potential fixes:



```
textBox2.Text = "error";
```

Generate field 'textBox2' in 'Form1'
Generate read-only field 'Form1.textBox2'
Generate property 'Form1.textBox2'
Generate local 'textBox2'
Generate class for 'textBox2' in 'Loops' (in new file)
Generate class for 'textBox2' in 'Loops'
Generate class for 'textBox2' in 'Form1'
Generate new type...

Some of these explanations are not terribly helpful, however!

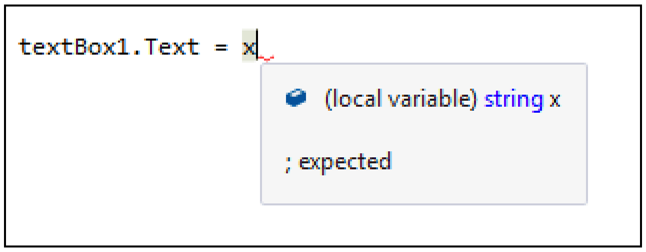
Red Marks

These are Syntax errors. (Syntax is the "grammar" of a programming language, all those curly brackets and semicolons. Think of a Syntax error as the equivalent of programming spelling mistake.)

In the code below, we've missed out the semicolon at the end of the line:

```
textBox1.Text = x
```

Holding the mouse pointer over the red wavy line gives the following message:



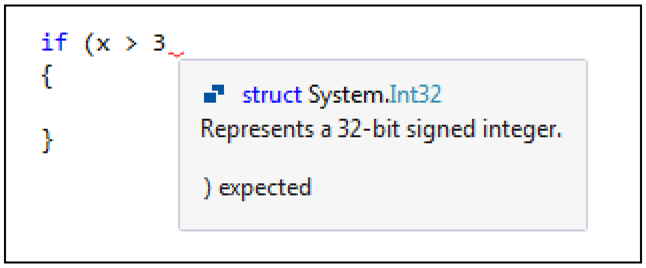
```
textBox1.Text = x
```

(local variable) string x

; expected

It's telling us that a semicolon (;) is expected where the red wavy underline is.

In the next image, we've missed out a round bracket for the IF Statement:



```
if (x > 3
{

}
```

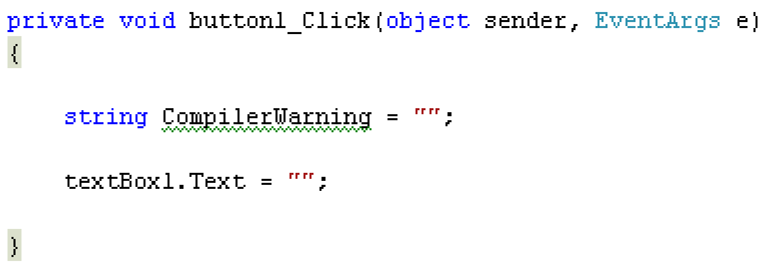
struct System.Int32
Represents a 32-bit signed integer.

) expected

Adding the round bracket will make the red wavy underline go away.

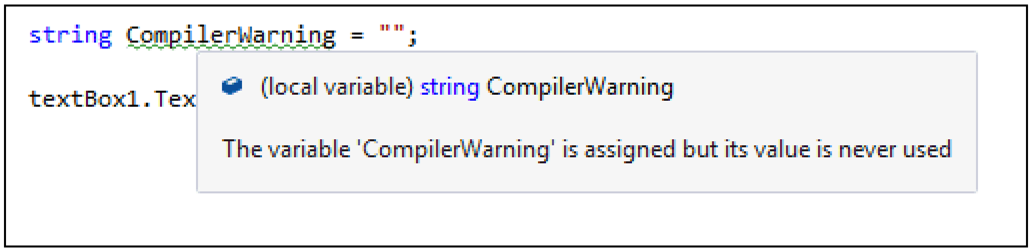
Green Wavy Lines

These are Compiler Warnings, the C# way of alerting you to potential problems. As an example, here's some code that has a green wavy underline:



```
private void button1_Click(object sender, EventArgs e)
{

    string CompilerWarning = "";

    textBox1.Text = "";

}
```

Holding the mouse pointer over the green underlines gives the following message:



```
string CompilerWarning = "";

textBox1.Tex
```

(local variable) string CompilerWarning

The variable 'CompilerWarning' is assigned but its value is never used

C# is flagging this because we have set aside some memory for the variable, but we're not doing anything with it.

This one is easy enough to solve, but some Compiler Errors can be a bit of a nuisance, and the messages not nearly as helpful as the one above!

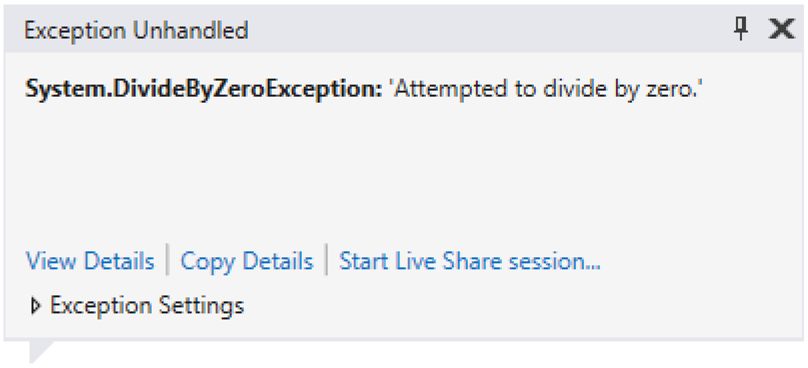
Whatever the colour of the underline, though, the point to bear in mind is this: C# thinks it has spotted an error in your code. It's up to you to correct it!

Run Time Errors In C# .Net

Run-Time errors are ones that crash your programme. The programme itself generally starts up OK. It's when you try to do something that the error surfaces. A common Run-Time error is trying to divide by zero. In the code below, we're trying to do just that:

```
private void button1_Click(object sender, EventArgs e)
{

    int Num1 = 10;
    int Num2 = 0;
    int answer;

    answer = Num1 / Num2;

}
```

The program itself reports no problems when it is started up, and there are no coloured wavy lines. When we click the button, however, we get the following error message:



Exception Unhandled

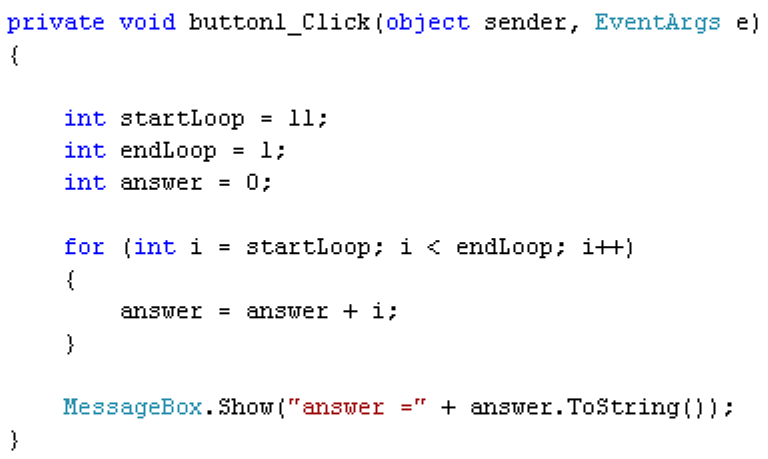
System.DivideByZeroException: 'Attempted to divide by zero.'

Had we left this in a real programme, it would just crash altogether ("bug out"). But if you see any error message like this one, it's usually a Run-Time error.

Look out for these type of error messages. It does take a bit of experience to work out what they mean; but some, like the one above, are quite straightforward.

Logic Errors In C# .Net

Logic errors are ones where you don't get the result you were expecting. You won't see any coloured wavy lines, and the programme generally won't "bug out" on you. In other words, you've made an error in your programming logic. As an example, take a look at the following code, which is attempting to add up the numbers one to ten:



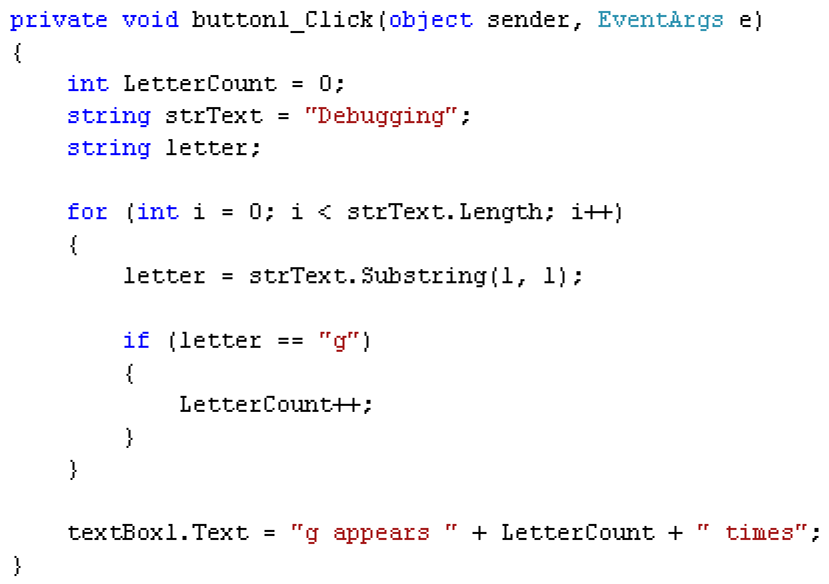
```
private void button1_Click(object sender, EventArgs e)
{

    int startLoop = 11;
    int endLoop = 1;
    int answer = 0;

    for (int i = startLoop; i < endLoop; i++)
    {
        answer = answer + i;
    }

    MessageBox.Show("answer =" + answer.ToString());
}
```

When the code is run, however, it gives an answer of zero. The programme runs OK, and didn't produce any error message or wavy lines. It's just not the correct answer!

The problem is that we've made an error in our logic. The **startLoop** variable should be 1 and the **endLoop** variable 11. We've got it the other way round, in the code. So the loop never executes.

Logic errors can be very difficult to track down. To help you find where the problem is, C# has some very useful tools you can use. To demonstrate these tools, here's a new programming problem. We're trying to write a programme that counts how many times the letter "g" appears in the word "debugging".

Start a new C# Windows Application. Add a button and a textbox to your form. Double click the button, and then add the following code:



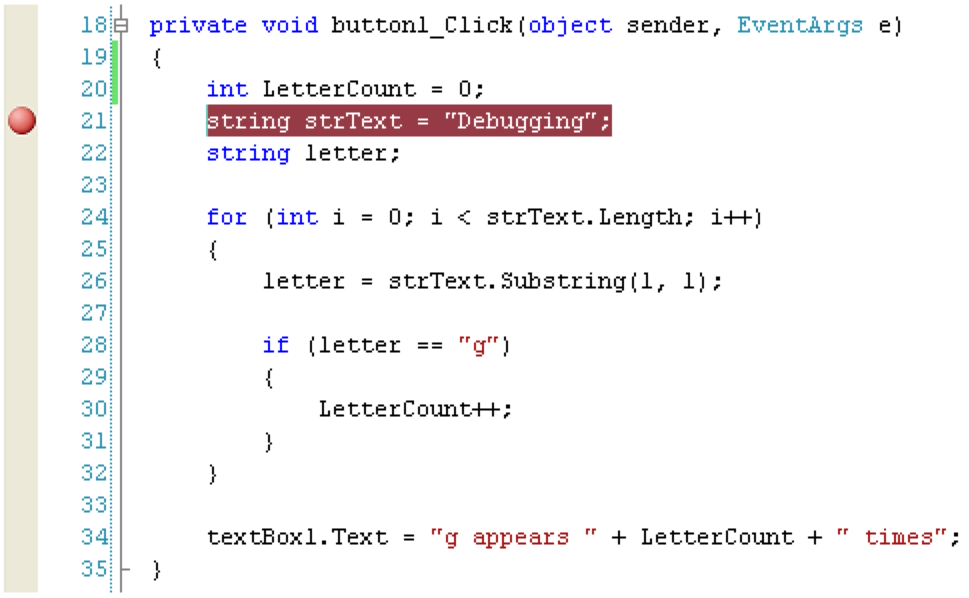
```
private void button1_Click(object sender, EventArgs e)
{
    int LetterCount = 0;
    string strText = "Debugging";
    string letter;

    for (int i = 0; i < strText.Length; i++)
    {
        letter = strText.Substring(1, 1);

        if (letter == "g")
        {
            LetterCount++;
        }
    }

    textBox1.Text = "g appears " + LetterCount + " times";
}
```

The answer should, of course, be 3. Our programme insists, however, that the answer is zero. It's telling us that there aren't and g's in Debugging. So we have made a logic error, but where?

Breakpoints In C# .Net

The first debugging tool we'll look at is the Breakpoint. This is where you tell C# to halt your code, so that you can examine what is in your variables. They are easy enough to add.

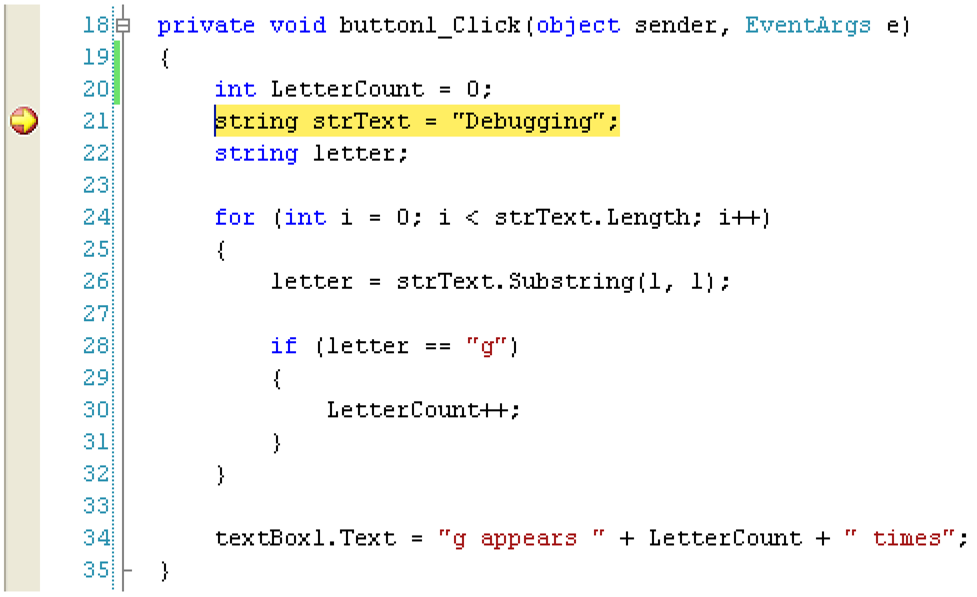
To add a Breakpoint, all you need to do is to click in the margins to the left of a line of code:



```
18  private void button1_Click(object sender, EventArgs e)
19  {
20      int LetterCount = 0;
21      string strText = "Debugging";
22      string letter;
23
24      for (int i = 0; i < strText.Length; i++)
25      {
26          letter = strText.Substring(1, 1);
27
28          if (letter == "g")
29          {
30              LetterCount++;
31          }
32      }
33
34      textBox1.Text = "g appears " + LetterCount + " times";
35  }
```

In the image above, we clicked in the margins, just to the left of line 21. A reddish circle appears. Notice too that the code on the line itself gets highlighted.

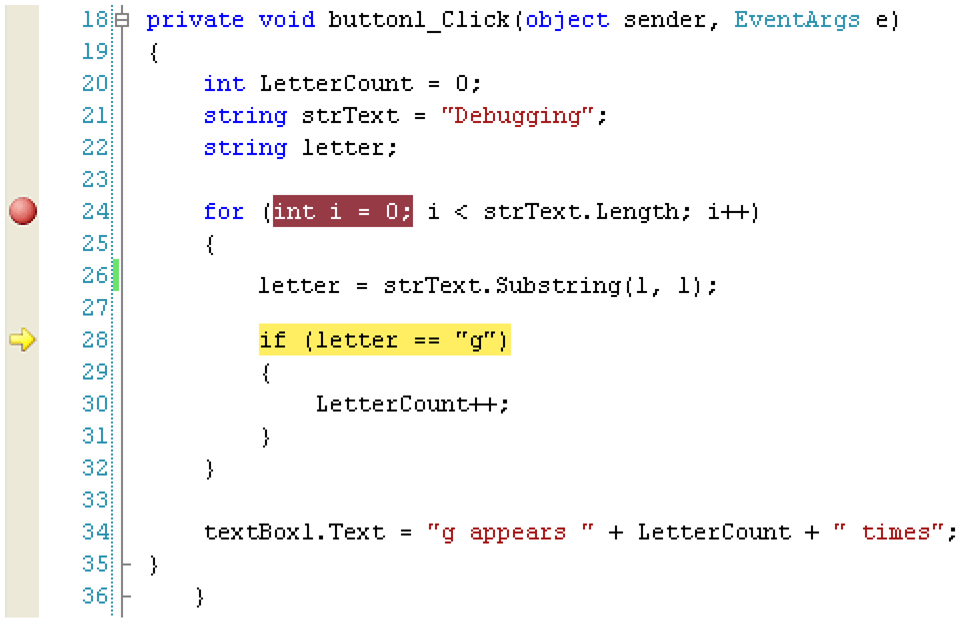
To see what a breakpoint does, run your programme and then click your button. C# will display your code:



```
18  private void button1_Click(object sender, EventArgs e)
19  {
20      int LetterCount = 0;
21      string strText = "Debugging";
22      string letter;
23
24      for (int i = 0; i < strText.Length; i++)
25      {
26          letter = strText.Substring(1, 1);
27
28          if (letter == "g")
29          {
30              LetterCount++;
31          }
32      }
33
34      textBox1.Text = "g appears " + LetterCount + " times";
35  }
```

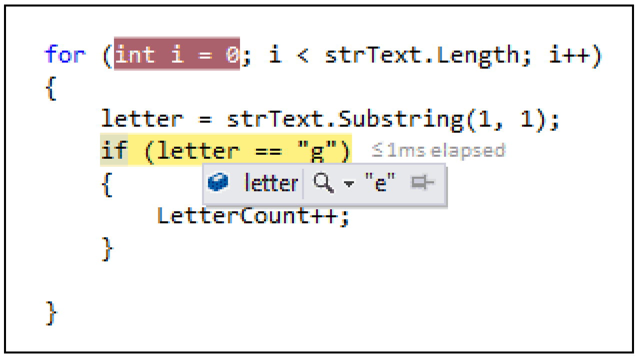
There will be a yellow arrow on top of your red circle, and the line of code will now be highlighted in yellow. (If you want to enable line numbers in your own code, click **Tools > Options** from the C# menus at the top. On the Options box, click the arrow symbol next to **Text Editor**, then C#. Click on **General**. On the right hand side, check the box for Line Numbers, under the **Settings** heading.)

Press F10 on your keyboard and the yellow arrow will jump down one line. Keep pressing F10 until line 28 in your code is highlighted in yellow, as in the image below:



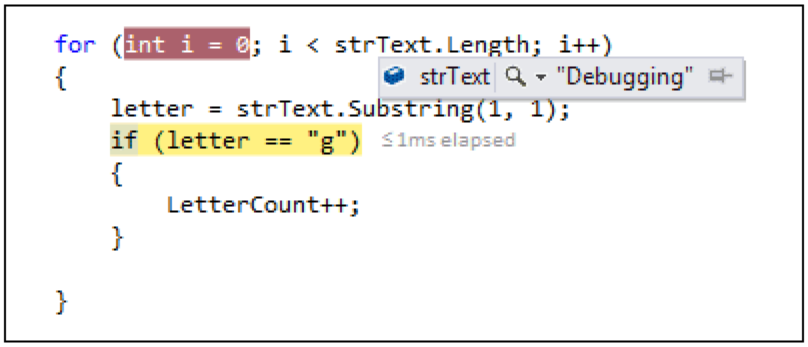
```
18  private void button1_Click(object sender, EventArgs e)
19  {
20      int LetterCount = 0;
21      string strText = "Debugging";
22      string letter;
23
24      for (int i = 0; i < strText.Length; i++)
25      {
26          letter = strText.Substring(1, 1);
27
28          if (letter == "g")
29          {
30              LetterCount++;
31          }
32      }
33
34      textBox1.Text = "g appears " + LetterCount + " times";
35  }
36      }
```

Move your mouse pointer over the **letter** variable and C# will show you what is currently in this variable:



```
for (int i = 0; i < strText.Length; i++)
{
    letter = strText.Substring(1, 1);
    if (letter == "g")  ≤1ms elapsed
    {        letter  "e"
        LetterCount++;
    }

}
```

Now hold your mouse over **strText** to see what is in this variable:



```
for (int i = 0; i < strText.Length; i++)
{
    letter = strText.Substring(1, 1);
    if (letter == "g")  ≤1ms elapsed
    {
        LetterCount++;
    }

}
```

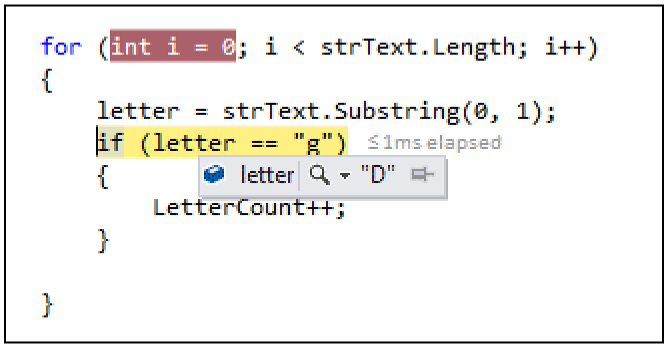
Although we haven't yet mentioned anything about the Substring method, what it does is to grab characters from text. The first 1 in between the round brackets means start at letter 1 in the text; the second 1 means grab 1 character. Starting at letter 1, and grabbing 1 character from the word Debugging, will get you the letter "D". At least, that's what we hoped would happen!

Unfortunately, it's grabbing the letter "e", and not the letter "D". The problem is that the Substring method starts counting from zero, and not 1.

Halt your programme and return to the code. Change your Substring line to this:

letter = strText.Substring(0, 1);

So type a zero as the first number of Substring instead of a 1. Now run your code again:



```
for (int i = 0; i < strText.Length; i++)
{
    letter = strText.Substring(0, 1);
    if (letter == "g")  ≤1ms elapsed
    {
        LetterCount++;
    }

}
```

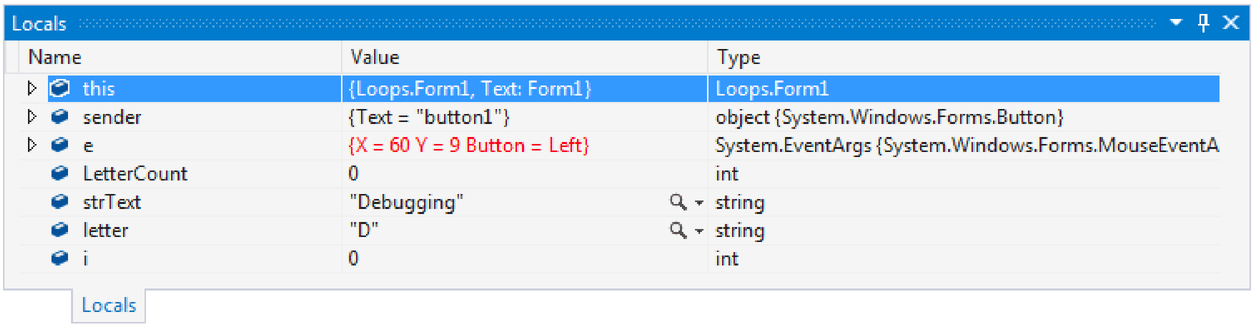
This time, the correct letter is in the variable. Halt your programme again. Click your Breakpoint and it will disappear. Run the programme once more and it will run as it should, without breaking.

So have we solved the problem? Is the programme counting the letter g's correctly?

The Locals Window In C# .Net

The Locals Window keeps track of what is in local variables (variables you've set up in this chunk of code, and not outside it).

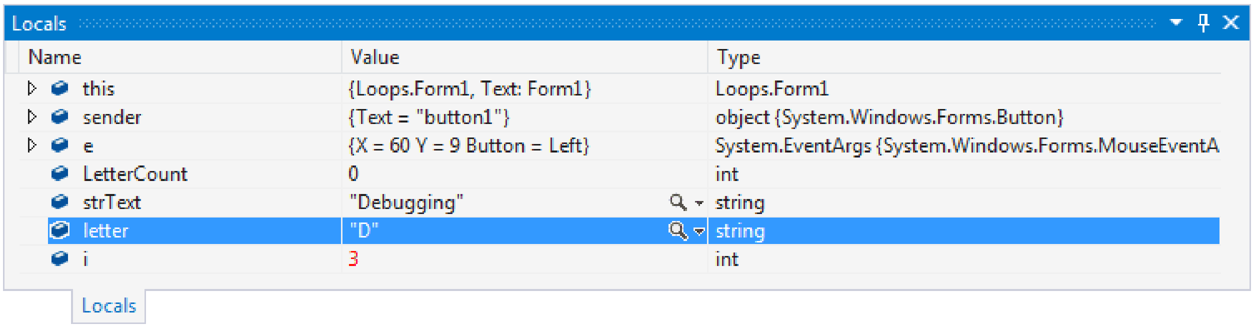
Add a new breakpoint, this time in the margins to the left of your IF statement. Run your programme again, and click the button. When you see the yellow highlighted line, click the Debug menu at the top of C#. From the Debug menu, click **Windows > Locals**. You should see the following window appear at the bottom of your screen:


Locals

Name	Value	Type
this	{Loops.Form1, Text: Form1}	Loops.Form1
sender	{Text = "button1"}	object {System.Windows.Forms.Button}
e	{X = 60 Y = 9 Button = Left}	System.EventArgs {System.Windows.Forms.MouseEventA
LetterCount	0	int
strText	"Debugging"	string
letter	"D"	string
i	0	int

Locals

Keep pressing F10 and the values will change. Here's what is inside of the variables after a few spins round the loop:


Locals

Name	Value	Type
this	{Loops.Form1, Text: Form1}	Loops.Form1
sender	{Text = "button1"}	object {System.Windows.Forms.Button}
e	{X = 60 Y = 9 Button = Left}	System.EventArgs {System.Windows.Forms.MouseEventA
LetterCount	0	int
strText	"Debugging"	string
letter	"D"	string
i	3	int

Locals

The variable **i** is now 3; **letter** is still "D", and **LetterCount** is still 0. Keep pressing F10 and go round the loop a few times. What do you notice? Keep your eye on what changes in your Locals window. The changes should turn red.

You should notice that the value in **i** changes but **letter** never moves on. It is "D" all the time. And that's why **LetterCount** never gets beyond 0. But why does it never move on?

Exercise I
Why does **LetterCount** never gets beyond 0? Correct the code so that your textbox displays the correct answer of 3 when the programme is run. HINT: think Substring and loops!

Try ... Catch In C# .Net

C# has some inbuilt objects you can use to deal with any potential errors in your code. You tell C# to **Try** some code, and if can't do anything with it you can **Catch** the errors. Here's the syntax:

```
try
{

}
catch
{

}
```

In the code below, we're trying to load a text file into a RichTextBox called **rtb**:

```
try
{

    rtb.LoadFile("C:\\test.txt");
}
catch (System.Exception excep)
{

     MessageBox.Show(excep.Message);

}
```

First, note the extra backslash after C:

C:\\test.txt

The extra backslash is called an escape character. You need to escape a single backslash because C# doesn't like you typing just one of them.

The code we want to execute goes between the curly brackets of try. We know that files go missing, however, and want to trap this "File not Found" error. We can do that in the catch part. Note what goes between the round brackets after catch:

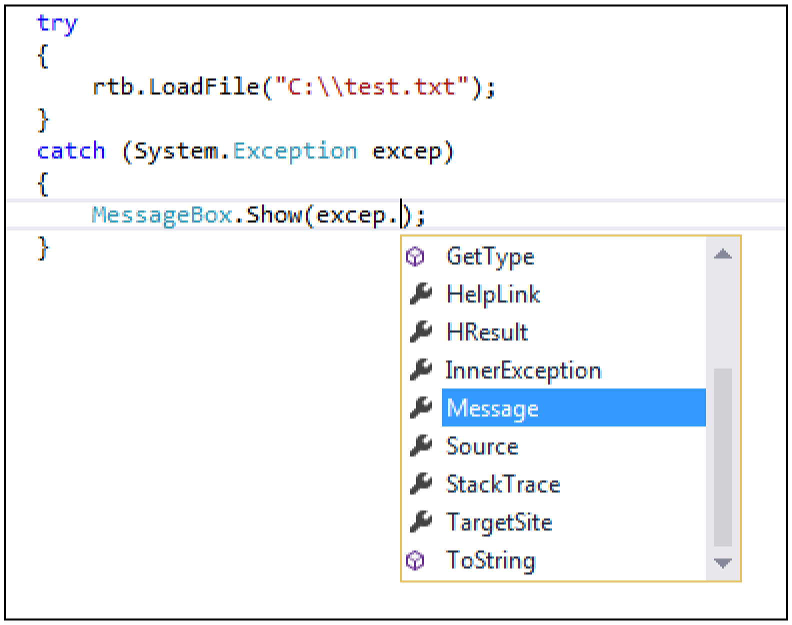
System.**Exception** excep

Exception is the inbuilt object that handles errors, and this follows the word **System** (known as a namespace). After **System.Exception**, you type a space. After the space, you need the name of a variable (**excep** is just a variable name we made up and, like all variable names, you call it just about anything you like).

The code between the curly brackets of catch is this:

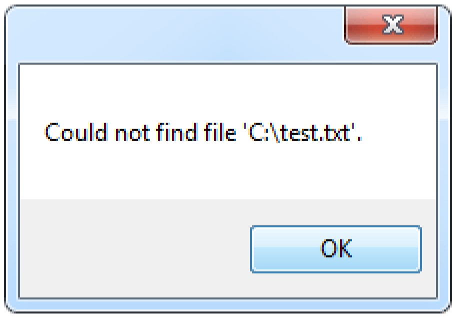
MessageBox.Show(**excep.Message**);

So we're using a message box to display the error. After the variable name (excep for us), type a dot and you'll see the IntelliSense list appear:



```
try
{
    rtb.LoadFile("C:\\test.txt");
}
catch (System.Exception excep)
{
    MessageBox.Show(excep.);
}
```

If you just want to display the inbuilt system message, select **Message** from the list. When the programme is run, you'll see a message box like this:



Could not find file 'C:\test.txt'.

If you know the type of error that will be generated, you can use that instead:

```
catch (System.IO.FileNotFoundException)
{

    MessageBox.Show("File not found");

}
```

To find out what type of error will be generated, use this:

```
          catch (System.Exception excep)
          {

                    MessageBox.Show( excep.GetType().ToString() );

}
```

The message box will tell you what system error is being generated. You can then use this between the round brackets of catch.

If you want to keep things really simple, though, you can miss out the round brackets after catch. In the code below, we're just creating our own error messages:

```
          try
          {

               rtb.LoadFile("C:\\test.txt");

          }
          catch
          {

               MessageBox.Show("An error occurred");

          }
```

You can add more catch parts, if you want:

```
          try
          {

               rtb.LoadFile("C:\\test.txt");

          }
          catch
          {

               MessageBox.Show("An error occurred");
```

```
}
catch
{

     MessageBox.Show("Couldn't find the file");

}
catch
{

     MessageBox.Show("Or maybe it was something
     else!");

}
```

The reason you would do so, however, is if you think more than one error may be possible. What if the file could be found but it can't be loaded into a RichTextBox? In which case, have two catch blocks, one for each possibility.

There's also a Finally part you can add on the end:

```
try
{

          rtb.LoadFile("C:\\test.txt");

}
catch (System.Exception excep)
{

          MessageBox.Show(excep.Message);

}
finally {

     //CLEAN UP CODE HERE

}
```

You use a **Finally** block to clean up. (For example, you've opened up a file that needs to be closed.) A Finally block will always get executed, whereas only one catch will.

(NOTE: there is also a **throw** part to the **catch** blocks for when you want a more specific error, want to throw the error back to C#, or you just want to raise an error without using try … catch. Try not to worry about throw.)

We won't be using **Try … Catch** block too much throughout this book, however, because they tend to get in the way of the explanations. But you should try and use them in your code as much as possible, especially if you suspect a particular error may crash your programme.