Selection the If Statement Try Catch and Validation

The main statement used in C# for making decisions depending on different conditions is called the *If statement. A second useful structure in a similar vein is called Try Catch.*

Using the IF statement, we may perform certain actions, only if a specific set of conditions are true.

The If Statement

The If-statement takes the following basic form:

If (*condition)*
{
'execute the code here
}

In the above example, if the condition is true, then the code between contained within {} will execute, otherwise, the code will be ignored.

For example...

```
//var to store the area code
string  AreaCode;
//var to store the city
string City = "";
AreaCode = "0116";
//if the area code is 0115
if (AreaCode == "0115")
{
    //set the city to Nottingham
    City = "Nottingham";
}
//if the area code is 0116
if (AreaCode == "0116")
{
    //set the city to Leicester
    City = "Leicester";
}
```

The value of the variable “City” would be set to “Leicester”.

else

Consider the following code...

```
//var to store the age
Int32 Age;
//copy the data from the presentation layer
Age =Convert.ToInt32( txtAge.Text);
//if the age is greater than or equal to 18
if (Age >= 18)
{
    //display a message saying age is ok
    lblMessage.Text = "You are old enough.";
}
else //otherwise
{
    //display a message saying age is not ok
    lblMessage.Text = "You are too young - go away!";
}
```

In the above code, if the age is greater than or equal to 18 then show a message "You are old enough".

Else, show a message "You are too young – go away".

Conditions

Notice the Boolean expression

Age>=18

We may use Boolean expressions to perform tests on our data using a range of operators.

We have relational operators that evaluate how values are related to each other, e.g.

>	a > b	is a greater than b
<	a < b	is a less than b
>=	a >= b	is a greater than or equal to b
<=	a <= b	is a less than or equal to b

We also have equality operators that test if two values are the same (or not).

For example

==	a == b	is a the same as b
!=	a != b	are a and b different

else if

There is another component of the If statement called Else If.

Look at the following code...

```
//var to store the age
Int32 Age;
//get the data from the presentation layer
Age =Convert.ToInt32( txtAge.Text);
//if the age is greater than or equal to 18 AND less than or equal to 30
if (Age >= 18 & Age <= 30)
{
    //ok
    lblMessage.Text = "You may come along";
}
    //else if the age is greater than 30
else if (Age > 30)
{
    //too old
    lblMessage.Text = "You are too old";
}
    //else if the age is less than 18
else if (Age < 18)
{
    //too young
    lblMessage.Text = "You are too young";
}
```

The idea is that we may perform a set of tests on the data and do different things under different conditions.

Notice the "and" logical operator & in the Boolean expression.

& is one of a range of logical operators that may also be used to evaluate data.

"And" may be used should we want to test if a value was greater than one value "And" less than another value.

Another useful operator is "*Or*" indicated by the symbol |

In which case, you might ask the question is a value one value "Or" another.

Try and Catch

Related to the If statement is another structure called Try and Catch.

Try Catch doesn't accept Boolean logic but tries to perform an operation. If that operation fails it runs the catch component.

```
try
{
    //try this
}
catch
{
    //if the thing above fails do this
}
```

This is related to the if statement in that both structures select different sections of code depending on certain conditions.

Validation

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs and the Universe trying to produce bigger and better idiots. So far, the Universe is winning."
Rich Cook

Validation is a very important aspect of our code design as by use of appropriate validation we will create programs that are both robust and responsive.

Robust means that a user may type any data they like into the program and it won't crash.

Responsive means that the program will tell the user what they have done wrong and what they should do to correct the error.

In looking at variables, you should have seen how selecting an incorrect data type for a variable will result in your program failing to compile.

For example

```
Int32 FirstName;
FirstName="Fred";
```

The Integer data types only accept whole numeric values within a certain range.

"Fred" is string (text) data and inappropriate for the Integer data type. For FirstName we should be using the String data type.

So, it is important that you select the right data type for your variables, however…

What about those "bigger and better idiots"? What about the users that will be using the software you create?

Entering data of the wrong type will crash your program.

If I have a field on my form called Age which only accepts numeric data what is to stop some idiot typing their name into it? What if instead of entering 14 they type "fourteen"?

Sending the wrong data to the correct data type will produce the same result as selecting the wrong data type in the first place.

To stop users entering inappropriate data, we need some system of checking the data entered.

The process of checking data as it is entered is called validation.

In looking at validation we shall introduce some of the validation functions but more importantly we see how the If statement may be used to validate our inputs.

We can use If to look at our data and decide if the data is useful/correct or not.

Testing for a Valid Number

Assume we have a field to capture a person's age on a form.

How can we check to see if the user has typed a suitable age value or not?

We want the logic to perform something like…

If the value entered by the user is a number
{
	Assign the data to our variable
}

Look at the following code...

```
//var to store the age
Int32 Age;
try
{
    //try this
    Age = Convert.ToInt32(txtAge.Text);
}
catch
{
    //if the thing above fails do this
    lblMessage.Text = "There was an error : the age is not a number";
}
```

Testing for a Valid Date

Similar code may be used to spot an invalid date.

```
//var to store the DOB
DateTime DateOfBirth;
try
{
    //try this
    DateOfBirth = Convert.ToDateTime(txtDOB.Text);
}
catch
{
    //if the thing above fails do this
    lblMessage.Text = "There was an error : the date of birth is not a valid date";
}
```

Testing for a Blank Field

The following code will do this…

```
//declare a variable to store first name
string FirstName;
//test the data typed in the text box to see if it isn't blank
if (txtFirstName.Text != "")
{
    //if it is ok read in the value
    FirstName = txtFirstName.Text;
}
else
{
    //if not ok then show an error
    lblMessage.Text = "You must enter your first name.";
}
```

Testing for Text over a Certain Number of Characters – the Length Method

Look at this code...

```
//declare a variable to store first name
string FirstName;
//test the data typed in the text box is not over 20 characters
if (txtFirstName.Text.Length <= 20)
{
    //    if it is ok read in the value
    FirstName = txtFirstName.Text;
}
else
{
    //    if not ok then show an error
    lblMessage.Text = "First name must not exceed 20 letters.";
}
```

Any items of string data type allow you to use the length method.

Validating an Email Address – The Contains Method

Contains is a useful method that scans the contents of a string to see if a character or sequence of characters is in that string.

If it can't find the characters it returns a value of false, if it does find it, it returns true.

```
// declare a variable to store the email address
string EMail;
//test the data typed in the text box has an @ symbol
if (txtEMail.Text.Contains("@"))
{
    //if it is ok read in the value
    EMail = txtEMail.Text;
}
else
{
    //if not ok then show an error
    lblMessage.Text = "Not a valid email address.";
}
```

Question – What is the problem with the above code?

Structuring your If Statements

Validation using if-statements can be quite a complicated process.

The problem with the above code is that the following are all valid email addresses.

fred@

@fred@

@

@@

@@@@@@@@

So, there must be additional validation required to check to see if the email is valid.

We need to think about what additional tests we need.

We could check for a . as in .com or .dmu.ac.uk.

We need to also check the length as the minimum (theoretical) length for an email address is five characters (though this would not happen in practice, the shortest I could find is name@v.gg!)

e.g.

a@a.a

When first constructing If statements it is a common mistake to do the following…

```
string EMail;                                   //declare a variable to store the email address
if (txtEMail.Text.Contains("@"))                //test the data typed in the text box has an @ symbol
{

    EMail = txtEMail.Text;                      //if it is ok read in the value
}
else
{
    lblMessage.Text = "Not a valid email address."; //if not ok then show an error
}
if (txtEMail.Text.Length > 5)                   //test the address is long enough
{
    EMail = txtEMail.Text;                      //if it is ok read in the value
}
else
{
    lblMessage.Text = "The address is too short.";  //if not ok then show an error
}
if (txtEMail.Text.Contains("."))                //test the data typed in the text box has an . symbol
{
    EMail = txtEMail.Text;                      //if it is ok read in the value
}
else
{
    lblMessage.Text = "Not a valid email address."; //    if not ok then show an error
}
```

What will happen when the following email address is processed by the code above?

fred@nothing

The email is read in by the first two If statements even though it isn't valid. By the time the third test processes the data the first two tests have already read it in!

One way to solve the problem is to nest the If statements like so…

```
string EMail;                                           //declare a variable to store the email address
if (txtEMail.Text.Contains("@"))                        //test the data typed in the text box has an @ symbol
{
    if (txtEMail.Text.Length > 5)                       //    test the address is long enough
    {
        if (txtEMail.Text.Contains("."))                //test the data typed in the text box has an . symbol
        {
            EMail = txtEMail.Text;                      //if it is ok read in the value
        }
        else
        {
            lblMessage.Text = "Not a valid email address."; //if not ok then show an error
        }
    }
    else
    {
        lblMessage.Text = "The address is too short.";      //if not ok then show an error
    }
}
else
{
    lblMessage.Text = "Not a valid email address no @ symbol.";//if not ok then show an error
}
```

If any test fails then the process stops and an error message is displayed.

Or we could take a very different approach which is not nested…

```
protected void btnSave_Click(object sender, EventArgs e)
{
    //var to store the email
    string EMail;
    //clear any previous errors
    lblError.Text = "";
    //if there is no @
    if (txtEMail.Text.Contains("@") == false)
    {
        //display an error
        lblError.Text = "You must include @ ";
    }
    //if email too short
    if (txtEMail.Text.Length <= 5)
    {
        //diaplsy an error concatenating with any previous errors
        lblError.Text =lblError.Text =  "The email must be more than 5 characters ";
    }
    //if there is no dot
    if (txtEMail.Text.Contains(".") == false)
    {
        //diaplsy an error concatenating with any previous errors
        lblError.Text = lblError.Text = "There must be a . ";
    }
    //if no errors read in the data
    if (lblError.Text == "")
    {
        EMail = txtEMail.Text;
    }
}
```

(Of course, this only tests the format of the email address not if the address actually exists or has been mistyped e.g. fred@dmu.ac.ku. As I said validation can be quite complicated.)