TREEVIEW

*This document gives information on the **Tree.sln** sample project. Note that this project contains many things that we've already looked at in this course – such as variables, functions or methods, overloaded methods and parameters. But there are also things that we have not yet encountered such as Enums and foreach loops. In addition, we necessarily need to look at details of the TreeView control and, later on, I use a fairly advanced (but important) programming technique called recursion – when a function calls itself. Don't worry about understanding everything in the code. For now, concentrate on the topics we've already covered to see how they are used in this project. We'll look again at this project later in the course.*

Any type of hierarchical data, from a family tree to a dusk file browser can be displayed and organised in a TreeView.

Let's see just how easy it is to create a tree view and add some branches (or 'nodes') to it. Start a new Windows Forms project and, from the Windows Forms Toolbox, drop onto the form a TreeView and a Button. Now double-click the button to add the `button1_Click` event-handler. To this event-handler, add these two lines of code:

```
TreeNode newNode = new TreeNode("[New Node]");
treeView1.Nodes.Add(newNode);
```

Run the application and repeatedly click the button. With each click a new branch is added at the same level as the original branch. We'll find out how to add indented branches in a moment. First, however, let's take a closer look at our code.

Note that we have created a TreeNode object called *newNode*. A TreeView maintains a collection of TreeNode objects. The Nodes property gives us access to this collection, an object of the TreeViewCollection class. A TreeViewCollection has properties and methods to manipulate its nodes. In the code above, we used the *Add()* method to add a new node. The TreeViewCollection class also provides methods such as *Clear()*, *Insert()* and *Remove()*.

Each node in a TreeViewCollection is an object of the TreeNode type. This object has a different set of methods and properties. For example, if you want to expand or collapse the nodes beneath a specific node, you must use the individual TreeNode's *Expand()* and *Collapse()* methods.

It is important to remember that a TreeView is made up of three different types of object: the TreeView object, the TreeViewCollection object (the TreeView's Nodes) and the TreeNode object (each node in the Nodes collection).

CHILD NODES

Now let's see how we can add indented or 'child' nodes. Load up the **Tree.sln** solution from the *Extras* folder of the source code archive. I have already created a few nodes. Run the application and click the *Add* button a few times. You will see that this creates a new node indented beneath the selected node. Try selecting different nodes and then clicking the *Add* button.

Turn to *AddBtn_Click()* in the code editor to see how I have programmed this. As before we start by creating a new TreeNode object, *newNode*. We then obtain a reference to the node that is currently selected in the *tv* TreeView:

```
TreeNode selNode = tv.SelectedNode;
```

Of course, there may be occasions when there is no node selected. This will be true, for example, when the first node is added. We therefore have to test if *selNode* is null. If it is, then we add a node to the Nodes collection and we select it using a property of TreeView:

```
tv.Nodes.Add(newNode);
tv.SelectedNode = newNode;
```

If, on the other hand, a node is selected, we add child nodes beneath it by adding to the selected node's *Nodes* collection rather than to the TreeView's *Nodes* collection. By default, the node will be 'collapsed' so any added nodes will be hidden beneath it. We therefore specifically cause it to expand itself to show the child nodes:

```
selNode.Nodes.Add(newNode);
tv.SelectedNode.Expand();
```

If you want to ensure that all the branches in the entire TreeView are expanded, use the *ExpandAll()* method. To ensure that all branches are collapsed, use *CollapseAll()*. Having added some nodes, you may want to be able to remove one or more at a later stage. Logically enough, the TreeNode's *Remove()* method does this. When you remove one node, any child nodes indented beneath it will also be removed. It is probably a good idea to warn the users about this, otherwise they might accidentally remove more data than they had intended.

You will find the node deletion code in the *DelBtn_Click()* method. The first thing we need to do before deleting a node is to check that the node exists. We do

this by checking that the selected node is not null. If it passes this test we then check it has any child nodes. To do this we simply count its *Nodes* property. If this is more than zero, then there are child nodes. Now we need to inform the user and ask if we should delete all the child nodes. The .NET framework supplies a *MessageBox* class for this job. The class is static so we don't have to create an object from it. The most important *MessageBox* method is *Show()*. This comes in no less than 21 overloaded variants!

> Remember, an overloaded method is one of a group of methods that share the same name – such as *Show()* but have different numbers or types of parameters such as: *Show(String), Show(String, String), Show(String, String, MessageBoxButtons)* and so on.

The version of *Show()* that use takes four arguments: the text string to be displayed, the caption string, the *MessageBoxButtons* and the *MessageBoxIcon*:

```
MessageBox.Show("There are branches beneath this node. Delete all?",
                "Warning!", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
```

MESSAGEBOXBUTTONS

MessageBoxButtons is an enumeration of constants (an *Enum*) that define the buttons in a message box.

> Enums are explained in step 8 of the course and Chapter 8 of The Little Book Of C#.

The available constants are: *AbortRetryIgnore, OK, OKCancel, RetryCancel, YesNo* and *YesNoCancel*. *MessageBoxIcon* is an enumeration that defines the icon to display. Its constants are: *Asterisk, Error, Exclamation, Hand, Information, None, Question, Stop* and *Warning*.

The *MessageBox* returns a *DialogResult* value according to the button that has been clicked. This again is an enumeration. its values are: *Abort, Cancel, Ignore, No, None, OK, Retry* and *Yes*. In my code, I declare a *DialogResult* variable, result, to which I assign the value returned from the *MessageBox*:

```
result = MessageBox.Show(
        "There are branches beneath this node. Delete all?",
        "Warning!",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question);
```

I then test the value of *result* and only if it is *DialogResult.Yes* (that is, if the user has clicked the 'Yes' button in the dialog) do I go ahead and delete the selected node and all its sub-nodes:

```
if (result == DialogResult.Yes) {
    tv.SelectedNode.Remove();
}
```

You can configure the appearance of the TreeView to a considerable degree simply by setting various Boolean (*true* or *false*) properties. For example, given a TreeView called *tv*, this is how you would cause it to display a line preceding the first or 'root' node, no lines connecting the other nodes and no plus or minus buttons to expand or collapse nodes:

```
tv.ShowPlusMinus = true;
tv.ShowLines = false;
tv.ShowRootLines = false;
```

Other properties let you toggle the display of check boxes before each node, enable or disable text editing of nodes and 'hot tracking' whereby each node is highlighted when the mouse passes over it. In my application, I assign values to these properties using check-boxes on the form. When the check box is checked its value is *true* otherwise it is *false*:

```
tv.ShowPlusMinus = PlusMinusCB.Checked;
tv.ShowLines = LinesCB.Checked;
tv.ShowRootLines = RootLinesCB.Checked;
tv.CheckBoxes = CheckBoxesCB.Checked;
tv.LabelEdit = LabelEditCB.Checked;
tv.HotTracking = HotTrackCB.Checked;
```

As you can see, we now have the makings of a pretty good TreeView application. But in order to be really useful, there are two very important things we need to do. First we need to be able to save the Tree to disk. And secondly we need to be able to reconstruct the Tree by reloading the saved data at a later stage. I'll return to look at this problem in *Step 7* of this course.

ITERATING THROUGH ALL THE TREE NODES

The way in which .NET organises a TreeView, with the Tree maintaining a collection of child nodes and each child node maintaining its own collection of other child nodes, may seem quite logical. All the same, it can be quite inconvenient to the

programmer. In particular, it can make heavy weather of any operation that needs to access all nodes at all levels in the TreeView. This is because the *TreeNodeCollection* owned by each node only comprises the next level of child nodes. It does not include any nodes at other levels that are owned by those child nodes. Try this out for yourself. Load my **Tree.sln** solution. We have already added some nodes to the TreeView, *tv*. Leave these as they are but click the *'Expand All'* button to ensure that they are all visible. Now from the *File* menu click the *Test* menu item. This uses the *Nodes.Count* properties of the TreeView and the *SelectedNode* to display the number of nodes owned by each.

You may be surprised to see that the TreeView only owns two nodes. This is because there are just two top-level nodes (here labelled *node0* and *node9*) these are the only 'child' node of the TreeView itself. Any selected node only owns the nodes at the level immediately beneath it. This being so, how can we work with all nodes at all levels? One simple way to get a count of all sub-nodes (the nodes indented at all levels) is to use the *GetNodeCount()* method. However, there may be times when we need more precise node-by-node control. Let's see how this might be done in code.

Look at the code in *IndexMI_Click()*. This executes when the *Index Tree* menu item is clicked. It uses a *foreach* loop to iterate through all the nodes owned by the TreeView.

> A `foreach` loop iterates through each element in a collection. We'll look at this more closely in Step 6.

Each node is passed as an argument to our *TraverseNodes()* method. This also has a *foreach* loop which calls the *TraverseNodes()* method, passing to it the current node. If you are unfamiliar with recursive methods, this may seem a little baffling at first sight. But see the explanation of '*Recursion*' down below. The *TraverseNodes()* method increments two values with each recursion. *NodeIndex* is declared outside the method and so its incremented value remains as the recursion unwinds. *NodeLevel* is incremented inside the method and so 'unwinds' with the recursion to decrement as well as increment node levels.

RECURSION

Or: *HOW LONG IS A PIECE OF STRING?*

When a method calls *itself* (as the *TraverseNodes()* method does in my example), it is said to recurse. Think of the entry point (the method heading) of a recursive procedure as a hole into a black box. Think of the data that goes into that hole as a piece of knotted string. When the method calls itself it feeds the string out through the bottom of the box and back in through the hole in the top. You end up with the string being repeatedly wound around and into the box. When no more data exists (e.g. when all the knots have been counted), the string starts unwinding itself. At the last turn, the string falls free of the box altogether. In coding terms , that is the point at which execution returns to the code that called the recursive method. In my *TraverseNodes()* method, the *int NodeLevel* is incremented whenever the method is called recursively as a new node is encountered. But when the string starts unwinding (when no more new nodes are found), the value of *NodeLevel* unwinds too, backtracking through its previous values. This is why the *NodeLevel* value decreases for higher-level nodes.

Recursion and other programming techniques and algorithms are beyond the core topic of this course in C# programming. The example in my code is kept deliberately simple and you do not need to understand recursion in order to complete this course. However, if you want to find out more about recursion, there are many online resources that explain it in more detail. Here are some you may find useful:

https://www.codeproject.com/Articles/142292/Recursive-methods-in-Csharp

http://www.c-sharpcorner.com/UploadFile/955025/C-Sharp-interview-questions-part4what-is-a-recursive-function-in/

https://msdn.microsoft.com/en-us/library/z3dk2cc3(v=vs.100).aspx

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Recursions/recursions.html

https://en.wikipedia.org/wiki/Recursion_(computer_science)