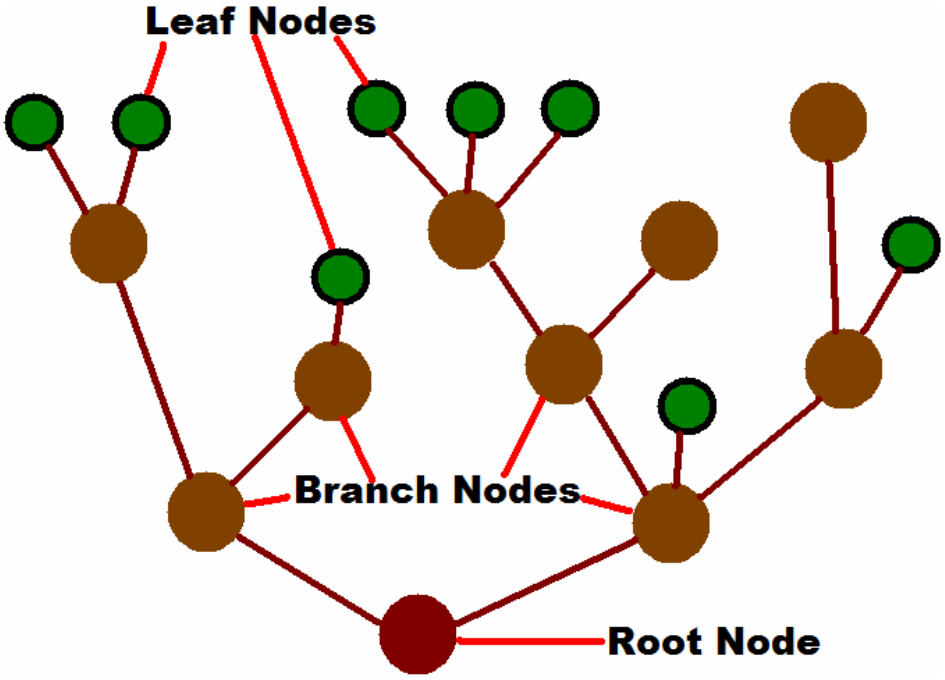
DynamicNodes:
Dynamic Branching Nodes
VB6 Version

A Very Fast and Extremely Powerful Node-Based System

by David Goben

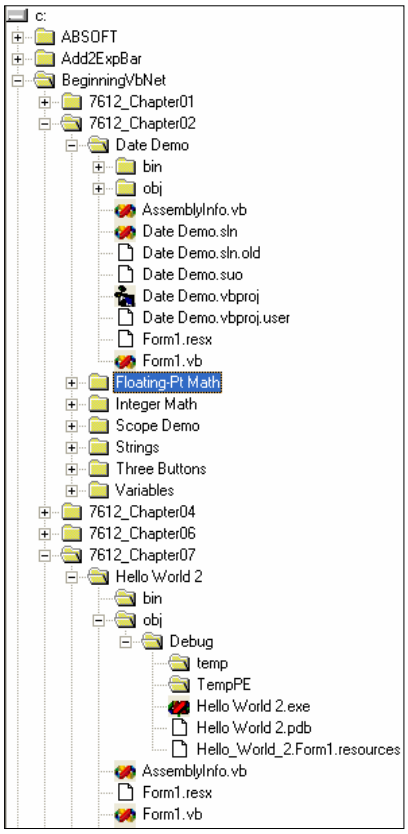


DynamicNodes: Dynamic Branching Nodes

Introduction

DynamicNodes is a powerful *ActiveX* library that supports a feature-rich hierarchical list of ***Node*** objects. Each node contains a rich set of properties and methods to support a wide variety of projects, from simply storing the headings of a document, the entries of an index, a dictionary, the files and directories of a disk, to being the core of a custom tree-view control.

DynamicNodes is capable of holding any information that can be stored in a branching, hierarchal structure. The provided DLL (or class files) supplies a primary **Node** class, called ***dynNode***, which defines the actual **Node** objects. It also supports a secondary class called ***dynNodes***, which is a ***Nodes Collection*** object automatically instantiated by the ***dynNode*** class whenever a new **Node** is created. This collection class is used to store references to potential sub-nodes for its associated **Node** object.



WHAT ARE DYNAMIC BRANCHING NODES?

Dynamic branching nodes defines a data structure that can be best exemplified by looking at the directory (folder) structure of a disk drive, such as the example on the right, where a directory is something that can contain a number of files, as well as other directories. In *Node* terms, each file and directory is considered a **Node**. A directory is simply a node that in turn contains (or can contain) other directories and/or files.

As you can see, ***Drive C:*** would be considered a ***Root Node*** (the *Root-Level Node* of a "tree" structure). Though in a disk directory a file cannot contain other files and other directories as a directory or subdirectory can, with nodes this is simply a matter of *perspective*, as any node is *potentially* capable of being the parent to any number of other nodes. Hence, you can look at a node as either a directory (a node that can or does contain other nodes), or as a file (a node that contains no other nodes and is not considered by your application to be a potential container).

Apart from directories, you can also look at a family tree, or breaking the structure of words down in phonemes or letters. A dictionary is such a structure, where the letter “A”, treated as a directory node, containing all the words (sub-nodes), in sorted order, beginning with the letter “A”.

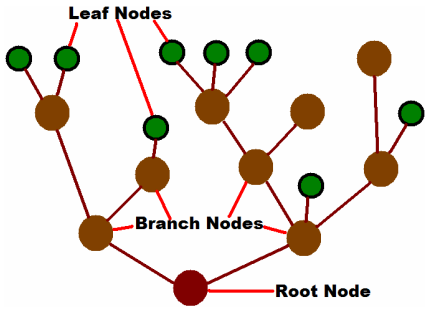
Unlike most node-based structures, ***DynamicNodes*** is not founded upon a list-based format with reference objects indexing into it – which also makes moving and removing nodes cumbersome and slow. Because of ***DynamicNodes'*** truly *dynamic* structure, it performs these, and all of its operations extremely fast and efficiently.

After creating a *Root-Level Node*, which is a node that acts as the root of a tree, you can create subordinate nodes that branch off from it, leading to other *Branches* (nodes that contain other nodes) or *Leaves* (nodes that do not contain other nodes). You can navigate through this tree by retrieving a reference to Node objects using ***Root***, ***Parent***, ***Child***, ***FirstSibling***, ***NextSibling***, ***PreviousSibling***, and ***LastSibling*** properties, as well as the ***FindID*** and ***FindKey*** methods, among many others, from the node's ***Node*** object. An ***Item*** property is also available from the Node's built-in ***Nodes*** *Collection* object.

A brief summary of some of the highlighted features of *DynamicNodes*:

- ***DynamicNodes*** *are perfect for tree-branching lists, binary lists, alphabetic lists, and anything else that catalogues its components into logical branching structures.*
- ***DynamicNodes*** *operates extremely fast, even on massive lists of many tens of thousands of nodes.*
- ***DynamicNodes*** *are not restricted to* ***32,000*** *nodes, as is a TreeView control, instead supporting potentially up to* ***2,147,483,648*** *nodes.*
- ***DynamicNodes*** *can be quickly accessed by reference, by Key, by ID, and by user-defined markers.*
- *Unlike the Nodes supported in a TreeView control,* ***DynamicNodes*** *have extensive developer support for node insertion, deletion, moving, marking, tagging, counting, and collecting. They offer a much more dynamic model to work from. User-definable properties allow you to easily expand the storage capacity of a node, featuring a traceable user-defined Boolean Marker, user-defined Object storage, and user-defined Variant storage, into which you can store any type variable, structure, or whatever you want!*
- *Fast sorting using the Shell-Metzner Sort, which is the fastest sort yet available, using the fewest number of swaps, and being geometrically faster than the standard Quick Sort.*
- ***DynamicNodes*** *offers you many features that are not available with standard nodes built into the VB environment. These features will all be covered in due course.*
- *Easy debugging for developers because errors can optionally be displayed in a Message Dialog Box if they enable this debugging property.*
- *This module provides all the workings to support a graphical tree-node-display control of your own design.*

Glossary: Understanding the Branching Node Lingo



There are a number of specialized terms used within this document (you have already run into a few of them), that you will need to understand in order to be on top of dealing with branching nodes. This list is in alphabetical order. Items that are also defined in the list are marked in **Bold**, so when you need a clearer understanding of the terms used in one description, you can refer to the definition of those words elsewhere in the glossary list.

Term	Definition
Ancestor	A **Node** that is the **Parent Node** of other **Nodes**, which **Descendant Nodes Branch** out from.
Branch	Every **Node** (**Leaf**) is a potential **Branch**. When it contains at least one **Child Node**, it is considered to branch like the limbs of a **Tree**, splitting off in different directions, and never intersecting with any other separate branches in the structure (see the branching diagram example, above).
Child	A **Node** that has a **Parent Node**. A Child **Node** is **Referenced** within a **Parent Node**'s ***Nodes collection*** object. It may also be referred to as a sub-**Component** of a **Parent Node**.
Children	More than one **Node** that share a common **Parent Node**. A groups of **Child Nodes** sharing the same **Parent** are considered to be **Siblings**.
Class	The defining structure or makeup of an **Object**. Publicly exposed **Members** of the Class are the **Methods** and **Properties** you work with to access a **Node** or its **Nodes Collection**.
Collection	A *container* that stores list of **Reference** variables to **Node Objects** that are subordinate, or **Children** of the **Node Object** storing the list.
Component	A **Member** of something. For example, a **Node** is a component of a **Tree**, and a **Method** and **Property** are components of a **Class**.
Declaration	When a reference variable for an **Object** is defined. Note that **Object Reference** variables can also have simultaneous Declaration and **Instantiation** when the **Reference** variable is declared with the *New* keyword.
Descendant	**Nodes** that are descended from a **Node**, sharing ultimately a single **Node** as an **Ancestor** (the **Root-Level Node**).
Function	A block of code that can be **Invoked** in a **Class** to perform a task, and returns a result value of a specified type. This is often referred to in Object Oriented lingo as a **Method**.
Hierarchy	Priority. The higher in the hierarchy, the closer one is to the **Root Node**. Hence, the **Root Node** has the highest priority.
Hierarchical	*See **Hierarchy.***
Initialization	When a newly created **Object** is set up for use. This involves providing it with characteristic through its **Properties** and **Methods**. Though this can be done at any time after an **Object** is created, often, as is the case of creating a **Root-Level Node** (covered shortly), it requires some initial information, such as *Text* and optional *Key* before it can actually be used.
Instance	A unique copy of an **Object**. The actual program code for the **Object** actually exists for all **Instances** just once, but the data (variable information) or each instance is set aside in the system **Resources** for each individual **Object**.
Instantiation	When an **Object** is created, such as with the *New* keyword.
Invoke	To call a **Subroutine Method** or **Function Method** in a **Class**. **Invoking** in object-oriented terms gives **Methods** an air of **Class** containment.
Invoking Node	The **Node** used to access a **Method** or **Property**.
Key	A unique identifier that makes it different from any other **Member** in either the entire **Tree** group, or among its **Siblings**.
Leaf	A **Node** without **Children**.
Lifetime	How long an **Object** lasts. The period between when an **Object** is created and when its resources are released (destroyed).
Member	A **Member** of a group. This is yet another **Reference** to a **Node**, which can be a **Leaf** (no **Child** nodes of its own), or a **Branch** (containing **Child Nodes**) or **Parent Node**. **Methods** and **Properties** are also considers Members of a **Class**.
Method	A **Subroutine** or **Function** that is a **Member** of a **Class**.
Node	A **Member** of a **Tree**; a **Leaf** of a **Branch**. Defined by the ***dynNode* Class**, this is an object that your code can manipulate in order to define it as a container or as a singular **Object** that is a **Child** of a container **Object**, or to contain other **Objects** (like a directory or folder).
Nodes Collection	Defined by the ***dynNodes*** class, it is a **Reference** list of **Nodes** that the associated node (***dynNode***) is the **Parent** of (a **Nodes Collection Object** is contained by and automatically created by every ***DynNode*** object).
Object	An **Instance** of a **Class**. Something that is considered to take up some system **Resources**.
Parent	A **Node** that contains **Child Nodes**. This can be looked upon as a **Branch** in terms of a **Tree**.

Property	A special **Method** that allows you to set (SET) **Objects**, assign values (LET), and/or retrieve (GET) information (value or **Object**) from a **Class**. Properties allow access to protected (normally inaccessible) **Members** of a **Class**. Primitive properties are simply declared as ordinary variables.
Reference	This normally is applied to variables that are used to access **Objects**. Unlike variables such as Integer, Long, and Single, an **Object** variable does not actually store the body of the **Object** itself, but is in fact a rather small item that simply *points* to the location where the **Object** actually exists. In many respects, this is more like String variables than you might think. Passing a String or Object variable by reference (ByRef) to a routine simply passes the small pointer data, but passing a String or Object by value (ByVal) passes the actual address of the **Object** itself. Hence, when you Set an **Object** to *Nothing*, it simply disables the Reference variable. The actual **Object** itself is not removed until either the environment does "garbage collection" and it finds that nothing, not even the **Nodes Collection**, still references it, or the application terminates. See your language reference guide for more information on *garbage collection* and **Object Lifetime**.
Resources	A portion of the computer's allocated memory space.
Root	This is the ultimate parent of a branching node system; a Node without a Parent Node. It is also called a **Root Node** and a **Root-Level Node**.
Root Level Node	*See **Root**.*
Root Node	*See **Root**.*
Sibling	Another node that shares the same parent node as the referenced node.
Subroutine	A block of code that can be **Invoked** in a **Class** to perform a take. This is often referred to as a **Method**.
Tag	A user-defined variant variable that can contain anything the user desires for it to contain (it is not used by the **Node Class**).
Tree	A description of the branching data structure of an ordered node-based system of **Root**, branches, and leaves.
TreeView	An ActiveX **Component** of the Visual Studio library that provides a visible, folder-based display of a **Tree**.
Visibility	The range at which an **Object** can be accessed. A variable defined within a **Method** can be accessed only by **Members** of that **Method**. A variable defined in the **Declaration** section at the top of a Form can be accessed directly by all members of that form, and externally when the **Object** is declared as Public and the referring item references the form. An **Object** that is declared as Public within a Module is visible to all levels of the application that the Module is used in.

Special Notes Regarding the VB Prototypes listed for the methods and properties

An understanding of properties and methods (functions and subroutines) is essential for using this *ActiveX* DLL. At the risk of being redundant, a very brief overview of these prototypes and how to employ them in your code is listed below. The following examples assume that a class object, from which the methods and properties are to be invoked, is named ***MyNode***. Variable ***MyObject*** is considered to be an object reference variable, and ***MyVariable*** is assumed to be a variable of the appropriate type, and ***MyBooleanResult*** is considered to be a Boolean variable.

Class Method Prototypes

Subroutine:

Example:
```
Public Sub Init(Optional Key As String, Optional Text As String)
```
Invocation:
```
Call MyNode.Init("MyUniqueKey", "MyText")
```
Or
```
MyNode.Init "MyUniqueKey", "MyText"
```

Function:
Example: `Function Move(NewParent As dynNode) As Boolean`
Invocation: `MyBooleanResult = MyNode.Move(BrandNewParent)`
Or `Call MyNode.Move(BrandNewParent)`
Or `MyNode.Move BrandNewParent`

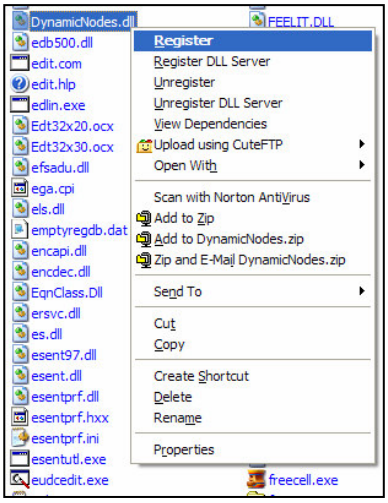
Class Property Prototypes

*Property **Set**:*
Example: `Property Set UserVar(UsrObject As Object)`
Invocation: `Set MyNode.UserVar = MyObject` `'object storage`

*Property **Let**:*
Example: `Property Let UserVar(UsrVariant As Variant)`
Invocation: `MyNode.UserVar = MyVariable` `'non-object variable storage`

*Property **Get**:*
Example: `Property Get UserVar() As Variant`
Invocation: `Set myObject = MyNode.UserVar` `'object storage`
Or `MyVariable = MyNode.UserVar` `'non-object variable storage`

Installing the DynamicNodes.DLL



To manually install the DLL can be attacked using two separate methods, both involve first copying the DLL to your system folder location (in actual fact, it being a COM object, it can be loaded to *anywhere* and registered, though this is not normally recommended).

First, copy the **DynamicNodes.DLL** file to your *System32* folder under the *Windows* Folder (C:\Windows\System32), or your *System* folder under *Windows* on ***Windows 95/98/ME*** systems (C:\Windows\System). Note that some systems may actually name their Windows folder by another name, such as *WinNT*, or some such. It is your system, and you should be familiar with it (if you are at a total loss, you really need to sit down with your system and get to know it better).

Next, you need to register the DLL using the *RegSvr32.Exe* utility, which has the syntax ***regsvr32 DllPath***. You can execute this by opening a DOS window to the appropriate system folder and enter the command. Or, right-click the DLL and select "***Open With***". If you see a "***Microsoft(C) Register Server***" listed, click that to register the DLL, otherwise select "***Choose Program...***" and search the system folder for *RegSvr32.exe*.

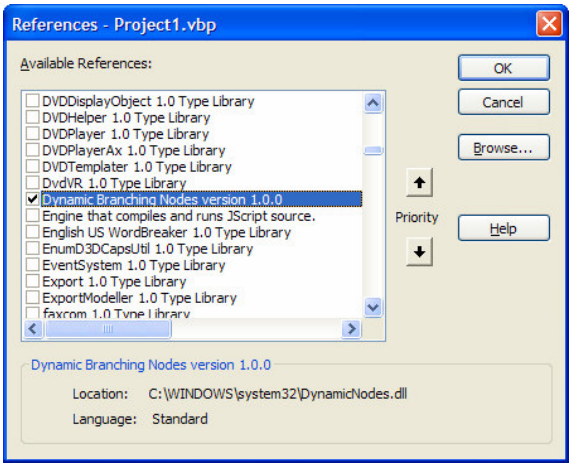
Personally, I have found the above registration methods to be a major pain in the rear end. Hence, an *much easier* method to register DLL's (and OCX files) is to employ a simple register/unregister command that you can quickly add to the *Windows Explorer's* popup menu. To add this to your system, which I most strongly recommend, you will also find this file in the Zip file contents named **RegUnreg.reg**. It contains the following text:

```
REGEDIT4
[HKEY_CLASSES_ROOT\.dll]
@="dllfile"
[HKEY_CLASSES_ROOT\dllfile\shell\Register\command]
@="RegSvr32.exe \"%1\""
[HKEY_CLASSES_ROOT\dllfile\shell\Unregister\command]
@="RegSvr32.exe /u \"%1\""
[HKEY_CLASSES_ROOT\.ocx]
@="ocxfile"
[HKEY_CLASSES_ROOT\ocxfile]
@="OCX"
[HKEY_CLASSES_ROOT\ocxfile\shell\Register\command]
@="RegSvr32.exe \"%1\""
[HKEY_CLASSES_ROOT\ocxfile\shell\Unregister\command]
@="RegSvr32.exe /u \"%1\""
```

Double-click the **RegUnreg.reg** file and answer *YES* to any prompts about making changes to the registry (this is a registry-modification file, so it makes sense – and ***no***, it does not do anything "scary"). Once installed, from *Windows Explorer* you can now right-click on a DLL or OCX file and you will see two new commands in the popup menu: ***Register*** and ***Unregister***. Right-click the **DynamicNodes.DLL** file and then select ***Register***. You are now ready to go off to the races with *Dynamic Branching Nodes*.

Note that non-COM (*Component Object Model*) DLL's will fail to register (if you try it while playing with the new functionality) by reporting that the required registration routine is missing or there is no registration helper. Well, non-COM DLL's do not need to be registered, so this is no big deal, and there is no harm done. Nothing was damaged, so relax.

Adding a Project Reference to DynamicNodes.DLL



Under the **Project / References...** menu of the VB6 IDE, you will find an entry in the list named ***Dynamic Branching Nodes version x.y.z***. Place a check into the checkbox next to it and select the **OK** button. You are now ready to take advantage of these *Dynamic Branching Node* objects. If for any reason you did not find this reference, then the ***DynamicNodes.dll*** file has either not be loaded onto your system, or it has not yet been registered using **RegSvr32.exe** – See the above section if you missed it. You can also ***Browse*** for it, and it will be automatically registered for you, if it was not.

Defining a *Root-Level Node*

Before getting into the detailed functionality of ***DynamicNodes*** (its *guts*), you should first consider how to initially define the most important part of a potential branching tree of nodes: the ***Root-Level Node***. These quick examples will show you that you have very little to fear about what some may believe to require procedures that might require special arcane knowledge and hard-to-memorize incantations. Indeed, you will be quick to discover that hierarchical node structures are very logical and orderly, and are therefore very easy to deal with programmically.

After ensuring that a reference to the *Dynamic Branching Nodes* object library is made for your project (see the previous page) you will need to create the root definition for your initial tree structure. Because there is no intermediate interface between you and ***DynamicNodes*** to do this for you behind the scenes as would be the case with a *TreeView* control, you will have to either *insert a simple line of initialization code* into your set-up routine, or else just as simply invoke the example ***NewNodeList()*** function supplied in the module file **moddynNodes.bas**. The code of this small method follows:

```
'*************************************************************************
' Function Name      : CreateNewRoot
' Purpose            : Return a new Root-Level Node to the invoker
'                    :
' Inputs             : Key: An optional text indentifier for the new node
'                    : Text: a name for the node (no unique requirement)
'                    :
' Outputs            : A new Root-Level Node
'                    :
' Assumes            : DynamicNodes.DLL is referenced in the project
'*************************************************************************
Public Function CreateNewRoot(Optional Key As String = vbNullString, _
                              Optional Text As String = vbNullString) As dynNode
  Dim cNd As New dynNode  'declare and instantiate a new Root-Level Node. STEP 1 of 2
  cNd.Init Key, Text      'initialize it (do this before anything else).  STEP 2 of 2
  Set CreateNewRoot = cNd 'return reference to created object (cNd is ALSO only a reference)
End Function
```

As you can see, as with any object system, there is object *declaration*, *instantiation*, and *initialization*. A last step of *releasing the resources* is of course, when you are finished using the object, to set it to **`Nothing`** (performing this task on a lower-level node, such as a *Root-Level Node*, will automatically in turn echo this task to all of its sub-ordinate nodes). Using the above VB function, you can create your root node thus:

```
'Place the following line of code in the heading of a module, or where-ever you plan to use it
Public RootNode As dynNode                'declare new Root-Level Node in a heading of a VB file
'Place the following line of code in the start-up portion of your program
Set RootNode = CreateNewRoot(Key, Text)   'instantiate and init new root node in a VB method
'.............................
'...Node-Use code goes here...
'.............................
'Place the following line of code in the termination portion of your program
Set RootNode = Nothing                     `release allocated resources at end of application
```

The "manual", do-it-yourself method requires only minor changes in the first two lines of code, and is in fact little different:

```
'Place the following line of code in the heading of a module,
' or where-ever you plan to use it
Public RootNode As New dynNode             'declare AND instantiate new Root-Level Node
'Place the following line of code in the start-up portion of your program
RootNode.Init Key, Text                    'initialize the new Root-Level Node ourselves
'.............................
'...Node-Use code goes here...
'.............................
'Place the following line of code in the termination portion of your program
Set RootNode = Nothing                      'release allocated resources at end
                                            ' of application
```

IMPORTANT REMINDER:

Because this is your base connection to the structure, it is certainly not something that you simply want to create and then toss away, because then you will not have any way to interface with the structure, and it will exist in limbo until the application that defined it closes. As such, it is a very good idea to maximize the visibility of this particular node in either a globally accessible location, such as being defined Public within a Module file, or better, expose it in a property definition within a class module, or restrict its visibility to only the level where you will be using it, such as Private within the heading of a form, class, or module.

A more elegant way to control *Root-Level Node* handling is to protect it as a private object within a small module, a form, or a class, which exposes an instantiation method, a *Get* property, and a release method. Possible code follows:

```
Option Explicit

'Private m_MyRootNode As dynNode 'A sample Root-Level Node object that is private
                                 'to this file

'************************************************************************
' Function Name     : DefineMyRootNode
' Purpose           : Define the Root Node object. This is a one-shot deal that
'                   : will only create the node if it does not exist
'************************************************************************
Public Function DefineMyRootNode(Optional Key As String, Optional Text As String) As
dynNode
  If Not m_MyRootNode Is Nothing Then 'if the object is already instantiated...
    Set m_MyRootNode = Nothing        'then release its resources
  End If
  Set m_MyRootNode = New dynNode      'now instantiate it...
  m_MyRootNode.Init Key, Text         'and initialize...
  Set DefineMyRootNode = m_MyRootNode 'return a reference to the Node to
                                      ' the caller, regardless
End Function

'************************************************************************
' Get Name          : MyRootNode
' Purpose           : Get the MyRootNode object
'************************************************************************
Public Property Get MyRootNode() As dynNode
  Set MyRootNode = m_MyRootNode
End Property

'************************************************************************
' Subroutine Name   : ReleaseMyRootNode
' Purpose           : Release the Root Node Object
'************************************************************************
Public Sub ReleaseMyRootNode()
  If Not m_MyRootNode Is Nothing Then   'if the object is instantiated...
    m_MyRootNode.Clear                  'optional. ensure child resources are cleared
immediately
    Set m_MyRootNode = Nothing          'release its resources and
  End If                                'those of all its children
End Sub
```

You may have noticed through all of this that the "***Key***" parameter is actually optional. If it is not supplied, then it will default to a unique ID *number* that is automatically generated for each object within the tree which is guaranteed not to repeat. Be aware also that *user*-defined Keys cannot *begin* with a numeric digit, as these "auto" keys are – this way we can be assured of their uniqueness. The "***Text***" portion is also optional, but there would be little use for most applications in that. Indeed, the only reason that it is tagged as optional is because it follows the optional Key definition the method declaration. Certainly, you can of course create nodes with blank Text fields. This is of course OK if you want to do that. But what is the point?

Also note that some developers prefer to optionally specify the *reference object* that the classes were defined in. In this case that would be ***DynamicNodes***. This has the advantages of the developer being able to quickly specify all publicly exposed classes, enumerations, etc. of a reference object without costing you anything extra in code (the compiler always resolves this with the exact same compiled code). Hence, declaring **RootNode** thus is also allowed: `Dim RootNode as DynamicNodes.dynNode`

You also need to consider the *lifetime* of the object and the requirements of its *visibility*. If you project need to access this object from different places in the application, you may wish to set up the definition statement for the root objects (you can have as many branching node collections as you require) in the declaration section of a form or module. Or, if you need to define it as ***Public*** in the declaration sections of the Form or Module. If it is needed throughout your application, and especially for the entire lifetime of your application or form, then in the application's termination code (such as the ***Form_Unload()*** event), you should ***Set*** the root node(s) to ***Nothing***. Although you can also instantiate the objects in a ***Form_Load()*** event, you do not really need to worry about it until you actually require it. However, in this case you should build a safety net under your code by handling the declaration something like this, where I am assuming that the Root-Level Node will be elsewhere declared as ***RootNode***:

```
If Not RootNode Is Nothing Then
  Set RootNode = NewNodeList(myKey, myText)
End If
```

Once a root node is defined, you create child nodes from it using its built-in ***Nodes*** *Collection* object, much like the *Nodes* collection in a *TreeView* control. The ***Nodes*** *Collection* contains an ***Add*** method with which you can invoke to add additional nodes, which also returns a reference of type ***dynNode*** to the code invoking this method.

A Brief Overview of dynNodes Class Properties (Nodes Collection)

Following is a table providing a quick reference to all exposed properties and methods of the ***dynNodes*** class that is exposed from within a ***dynNode*** object. More details will be provided, including property and method prototypes later on in this document:

Property	Type of Decl.	Return Type	Description
Add	Method	dynNode	Add a new ***Node*** to the node's ***Nodes*** *Collection*. Return a reference to the newly created node.
Clear	Method	————	Remove all child nodes (and any children they may have). This effectively cleans the entire branch out.
Count	Property Get	Long	Get the number of child nodes contained by the associated node. This is a count of only its IMMEDIATE children (those nodes who share the specified node as their parent).
InitAutoForNext	Method	Boolean	Initialize automatic For-Next processing of the **Item** property so that the user can scan through all items in a Nodes list without specifying a specific index into that list.
Item	Property Get	dynNode	(Default) Get a reference to the specified child items. This is the default property of the ***dynNodes*** class. Hence, you can specify a item in the *Nodes Collection* as either **ParentNode.Nodes.Item(Index)** or as **ParentNode.Nodes(Index)**. Index can be either a numeric sequence within the list of Nodes (1 to ***Nodes.Count***), or a Key, which is not confined to the Nodes Collection references, but can actually point to *any* Node in the tree.
Remove	Method	————	Removes the specified indexed item from the *Nodes Collection*.

dynNodes Details (Nodes Collection)

In order to provide practical examples of using the actual ***Node*** object, as defined by the ***dynNode*** class, we must first understand how to add nodes to them. In that case, we will first explore the ***dynNodes*** *Collection* object, which is subordinate to the ***dynNode*** object and is automatically created within each new ***dynNode*** object. In English, this means that you never have to sweat the instantiation details for the collection object, although you will often find yourself using its methods and properties.

Note that the ***dynNodes*** collection is a 1-based collection. Hence, its ***Count*** property is considered to also be its upper bounds.

As nodes are added to the collection, they are assigned a unique ID number (unique to the tree that it is a member of, anyway) that is stored in the ID property of the node. Unlike the *TreeView* control, a ***dynNode*** object is not assigned a fixed index value within the collection. This is due the sort-ability nature of these nodes, although it is certainly not like they just "float around" all over the place. You can define a reference variable of type ***dynNode*** to a node, add many references that will shift them around, especially with the ***Sorted*** property being set, but your node reference variable will *still* point to the *same* node (technically, Object Variables simply contain reference pointers to Objects that are actually stored *elsewhere*. You can shift these reference pointers around all you want, but the actual object data remains stationary without likewise being shifted around.

The ***Nodes*** object is access by specifying a ***Node*** object, adding a *dot*, and then the ***Nodes*** parameter. For example, if we assume that we are starting with the root ***Node*** object, which we will call ***RootNode*** for our example, we can access to its ***Nodes*** property by entering `RootNode.Nodes`, and then adding another *dot*, and following that will the method or property of the *Nodes Collection* that you wish to use.

Note further from the above table that the ***Item*** property is the *Default* property for ***dynNodes*** object. Being that ***Item*** specifies an indexed list of objects, you can access an indexed item in the ***dynNodes*** list by entering `RootNode.Nodes.Item(Index)` or as simply `RootNode.Nodes(Index)`, much like you can access members of a standard VB *Collection* object.

Add (Nodes Collection)

VB Prototype:

```
Public Function Add(Optional Relative As Variant = 0&, _
     Optional Relationship As dynNodeRelationship = dynNodeChild, _
     Optional Key As String = vbNullString, _
     Optional Text As String = vbNullString) As dynNode
```

The ***Add*** method allows you to add a Child or Sibling Node to the *Invoking Node*. This method expects fours parameters, all of which are optional (though this is normally not practical).

The ***Add*** method creates a new Node and adds it to the node tree, returning a reference to this new node. Where this new node reference is added is determined by a combination of the ***Relative*** and ***Relationship*** parameters.

Syntax

*Node.Nodes.***ADD***([Relative], [Relationship], [Key], [Text])*

Part	Description
Node	Required. An object expression that evaluates to a ***dynNode*** object. This is called the *Invoking Node*.
Nodes	The Node's *Node Collection* object. This object contains a potential list of node references that are considered to be child nodes of Node.
Relative	Optional. The index number or key of a pre-existing ***dynNode*** object. The relationship between the new node and the pre-existing node is found in the next argument. If Relative is 0 or not supplied, the specified object is considered to be the node from which this method was invoked (the *Invoking Node*).
Relationship	Optional. Specifies the relative placement of the new node object in relation to the specified object. See the Relationship table below for an explanation of the valid settings for this parameter. Default is **dynNodeChild**.
Key	Optional. A unique string that can be used with the **Add**, **Item**, and **Remove** methods.
Text	Optional. The text that is considered to be descriptive of the new node.

The ***Relative*** parameter is the *Index Value* or *Key* of a pre-existing Node object. If ***Relative*** is not supplied, or 0, then the current node from which this command was invoked is assumed. *e.g.*, ***MyNode*** in **`MyNode.Nodes.Add(, ...)`**. If an *Index Value* is supplied, it must be a valid range from 1 to the number of nodes contained within the collection of the *Invoking Node* (***MyNode*** in this example, will have a maximum range found in **`MyNode.Children`** or **`MyNode.Nodes.Count`**), and that child node will be referenced instead of the *Invoking Node*. If a *Key* is supplied, the Key's node will be referenced instead of the *Invoking Node*.

The ***Relationship*** parameter specifies whether it will be creating the new node as a *Child* of the referenced Node, or as a *Sibling* (contained within the same collection as the referenced node, and hence sharing the same parent node with it).

The ***Relationship*** parameter specifies the relative placement of the new Node object in regard to the specified node, as described in the following table:

dynNodeRelationship	Value	Meaning
dynNodeFirst	0	First. The new ***Node*** is placed before all other nodes at the same level of the node specified or implied in *Relative*.
dynNodeLast	1	Last. The new ***Node*** is placed after all other nodes at the same level of the node specified or implied in *Relative*. Any Node added subsequently may be placed after one added as Last.
dynNodeNext	2	Next. The new ***Node*** is placed after the node named in *Relative*.
dynNodePrevious	3	Previous. The new ***Node*** is placed before the node named in *Relative*.
dynNodeChild	4	Child. **Default.** The new ***Node*** becomes a child node of the node named in *Relative*.

The default ***Relationship*** parameter is ***dynChild***, which will make the new node a child of the specified node. If the specified node is actually a *Root-Level Node*, then you should *be sure* to use this ***dynNodeChild*** relationship, as a *Root-Level Node* is the *only* node in a tree structure that cannot have siblings, and so sibling-related relationships in regard to it will result in an error. All other Relationship options are considered to be *Sibling-Related*, because they specify a relationship at the same level as the specified node, sharing a common parent node with it.

If the new node's container node's ***Sorted*** property is set, then this actually makes ordinal insertions pointless, because the nodes will be sorted after the insertion. In this case, the sibling-related command of ***dynNodeLast*** will be assumed regardless of the actual sibling relationship that you specified because this is the fastest relationship to add.

Using the returned node reference from the ***Add()*** method, you can initialize the new node further to complete the design specification that you have set up for your application. However, it may be a really good idea to first check to see if the returned node is actually set to *Nothing*, which would indicate an error. In this case you can check the ***ErrorCode*** property of the *Invoking Node* to see what error was triggered (there are alternatives to this form of error trapping, which will later be discussed. See the ***Handling Error Conditions*** section).

The ***dynNodeChild*** options must be further clarified. It tells the ***Add()*** method to make the new Node a child of the specified node, rather than as a sibling. Hence, the following example will make the new node a child if the second child node of MyNode:

```
Dim NewNode As dynNode
Set NewNode = MyNode.Nodes.Add(2, dynNodeChild, "Inventory Sub-Total", _
                               "Inventory Sub-Total Heading")
```

Returns

A reference to the new ***dnyNode*** object. *Nothing* if there was an error. See also ***ErrorCode*** and ***ErrorMsgBox*** in the ***dynNode Details***.

In the following example, we are going to make the ***NewNode*** a child of the ***MyNode*** node. Note that all other ***Relationship*** commands deal with the specified node (***MyNode***) and its group of sibling nodes (sharing a common *Parent Node*):

```
Set NewNode = MyNode.Nodes.Add( , dynNodeChild, "Inventory Sub-Total", _
                                " Inventory Sub-Total Heading")
```

In the following example we will do the same thing with a ***DynNode*** reference to a child node of *MyNode* named *SiblingNode* (we could have also simply specified the actual index if we knew it, but this latter method is more prone to error if your code is not on top of things). Hence:

```
Set NewNode = MyNode.Nodes.Add(SiblingNode.Index, dynNodeChild, _
              "Inventory Sub-Total", " Inventory Sub-Total Heading")
```

or if we knew the desired Relative Node was the second child node of *MyNode*:

```
Set NewNode = MyNode.Nodes.Add(2, dynNodeChild, "Inventory Sub-Total", _
                                " Inventory Sub-Total Heading")
```

The ***Key*** parameter is a string of text that uniquely identifies this node against its all other nodes. If one is not supplied, it will default to a string rendition of the new node's ID value, which will guarantee it's ***Key*** uniqueness.

The ***Text*** property is something that might be considered as the 'viewable' property of the node. This text is what is displayed in a *tooltip* when you point to a node in the VB IDE during single-step debugging, due to the ***Text*** property being the *Default* property for ***dynNode*** objects.

Be aware than if you specify a ***Key*** instead of an *Index Value* for the ***Relative*** parameter, then the *Key (* the node "indexed" by Key) does not have to be a member of the *Nodes Collection* for ***MyNode***. This functionality allows you to reference <u>*any*</u> node in the *entire* tree using the current node reference as an invocation base. Although this requires some exacting understanding of your code and tree structure, where you have established a logical layyour format for defining key values (a directory path format is a useful suggestion) as opposed to the simpler numeric index which will confine itself to ***MyNode***'s *Nodes Collection*.

In case you haven't guessed, you can now use this new node reference to create additional child nodes from it. With what little code we have seen so far, we now have enough practical knowledge to create a structure that will create a node tree for the entire folder and file contents of a disc drive (which can be quite a lot! Currently, one of my personal systems has in excess of 300,000 folders and files contained on it). An example of such code is listed later in this document.

Add example:

```
Dim MyNode As dynNode

Set MyNode = RootNode.Nodes.Add(, dynNodeChild, MyKey, MyText) 'Add child to RootNode
If Not MyNode Is Nothing Then
  MyNode.Sorted = True  'Node will be sorted (this must be set for each "parent" node)
  MyNode.Marked = True  'user-def usage.  I am using it here to mark folders-type nodes
  MyNode.Tag = "Folder" 'user-def usage.  I am using this to describe the type of node
Else
  Msgbox "Error " & CStr(RootNode.ErrorCode) & _
         ". Description: " & RootNode.ErrorDescription(RootNode.ErrorCode)
End If
```

See Also*: DynNode Details*

NOTE: *This Method can generate trappable errors*

Clear (Nodes Collection)

VB Prototype: `Public Sub Clear()`

The ***Clear*** method removes all descendant nodes from the *Invoking Node's* list. All children, grandchildren, and any further descendents are released of their resources. This will be invoked automatically when the resources for the *Invoking Node* are cleared or set to *Nothing*, but this invocation ensures that it will happen immediately, as sometime an object's *Terminate* event is not fired until the application ends. This method is especially handy if you want to rebuild the branches or children of the *Invoking Node*, or redefine them.

Syntax

*Node.Nodes.***Clear**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Nodes is an object expression that resolves to a ***dynNodes*** object.

Returns

No return value.

Clear example:

```
Debug.Print MyNode.Text & " contains " & CStr(MyNode.Nodes.Count) & " sub-nodes"
MyNode.Nodes.Clear  'remove all child nodes.  The following line will report zero nodes
Debug.Print MyNode.Text & " contains " & CStr(MyNode.Nodes.Count) & " sub-nodes"
```

Count (Nodes Collection)

VB Prototype: `Public Property Get Count() As Long`

The ***Count*** property returns the number of nodes contained within the ***Nodes Collection*** for the *Invoking Node*. If zero, then the list is empty. This value can be used to loop through the ***Nodes Collection*** object to inspect its referenced nodes.

Syntax

*Node.Nodes.***Count**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Nodes is an object expression that resolves to a ***dynNodes*** object.

Returns

The number of nodes that are immediate children of this node.

Count example:

```
Dim Index As Long
Debug.Print MyNode.Text & " contains " & CStr(MyNode.Nodes.Count) & " sub-nodes"
For Index = 1 To Node.Nodes.Count           'Note that node-lists are 1-based
  Debug.Print "  " & Node.Nodes(Index).Text 'list the child node's text values
Next Index
```

InitForNext (Nodes Collection)

VB Prototype: `Public Function InitAutoForNext() As Boolean`

The ***InitForNext*** method Initializes automatic For-Next processing of ***Item*** properties so that the user can scan through all items in a *Nodes* list without specifying a specific index into the list. Supplying an Index of 0 (zero) will force the ***FirstSibling*** or ***NextSibling*** to be returned. If the returned **Node** is *Nothing*, then the list is exhausted. This method returns *True* if the operation was successful. If *False*, then the Node has not been initialized (note that this error will only occur on a *Root-Level Node* that has not been initialized (sub-nodes are auto-initialized)).

Syntax

*Node.Nodes.***InitForNext**

Returns

False if the node has not been initialize (this is only possible on an un-initialize *Root-Level Node*).

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Nodes is an object expression that resolves to a ***dynNodes*** object.

InitForNext Example:

```
Dim Node As dynNode


With RootNode.Nodes
  .InitAutoForNext                    'ensure we start at the beginning
  Do
    Set Node = RootNode.Nodes.Item()  'Grab an item from the list (note: default
Index=0)
    If Node Is Nothing Then Exit Do   'if nothing in list left to grab, then break out
of loop
    Debug.Print Node.Text             'show it's Text property
  Loop
End With
```

NOTE: *This Method can generate trappable errors*

Item (Nodes Collection) (DEFAULT Property)

VB Prototype:

```
Public Property Get Item(IndexOrKey As Variant) As dynNode
```

The ***Item*** property returns a node reference from the ***Nodes Collection*** at the specified index. Collections are 1-based, and so the ***Count*** property indicates the upper bounds of the collection, and **1** represents the lower bounds.

Syntax

*Node.Nodes[.***Item***](IndexOrKey)*

Returns

A reference to the indexed ***dnyNode*** object. *Nothing* if there was an error. See also ***ErrorCode*** and ***ErrorMsgBox***.

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Nodes is an object expression that resolves to a ***dynNodes*** object.

IndexOrKey is either an expression that results in an integral value that would be a sequential index within the collection of child nodes contained by the *Invoking Node*, or is a text string containing the unique Key of the desired Node within the Tree itself.

Item is the ***Default*** property for the ***Nodes Collection***, which means that the ***Item*** token is not required when referencing sub-items in the ***Nodes Collection***, as you may have seen in the ***Count*** property example, above. Indeed, the only time that you would need to explicitly specify the ***Item*** property is when you may be using the ***Nodes Collection*** in a ***With*** block. Here is the *same* example as shown for the ***Count*** property, but explicitly using the ***Item*** property:

```
Dim Index As Long
With Node.Nodes
  Debug.Print Node.Text & " contains " & CStr(.Count) & " sub-nodes"
  For Index = 1 To .Count                'Note that node-lists are 1-based
    Debug.Print "  " & .Item(Index).Text '(we cannot use the default property behavior
here)
  Next Index
End With
```

Like with a *TreeView* or a *Collection* object, you can also specify ***dynNodeIndex*** as a ***Key*** value (hence the Variant declaration). In this case, it's operation is exactly like the ***FindKey()*** method defined later for the ***dynNode*** object, in that it will look not within just the current Node's collection of sub-nodes, but will in fact search through the *entire* node list of the tree. Although using the ***FindKey()*** method from the Node object is cleaner, this enhanced functionality is included for those who have node-referencing using a *Key* ingrained in their brains.

See Also*: DynNode Details* **NOTE:** *This Property can generate trappable errors*

Remove (Nodes Collection)

VB Prototype: `Public Sub Remove(IndexOrKey As Variant)`

The ***Remove*** method allows you to remove an indexed item from the ***Nodes Collection*** and release its resources. The indexed value is from ***1*** to ***Nodes.Count***, or it is a Key which references the item to be removed from the tree. If the removed item contains children or further descendants, then they are all automatically removed as well.

Syntax
*Node.Nodes.***Remove***(IndexOrKey)*

Returns
No return value.

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Nodes is an object expression that resolves to a ***dynNodes*** object.

IndexOrKey is either an expression that results in an integral value that would be a sequential index within the collection of child nodes containing the Invoking Node, or is a text string containing the unique Key of the desired Node.

The following example of using the **Remove** method in a manner that is similar to the ***Clear*** method:

```
With Node.Nodes      'FLUSH the Nodes Collection
  Do While .Count    'while there are items to remove
    .Remove 1        'remove an item (higher items will 'collapse' down onto it)
  Loop               'flush entire collection
End With
```

Like with a *TreeView* or a *Collection* object, you can also specify ***IndexOrKey*** as a ***Key*** value (hence the Variant declaration). Note that in the case of specifying a Key instead of an index into the *Nodes Collection*, it will instead quickly search through the *entire* node list of the tree.

NOTE: *This Method can generate trappable errors*

A Brief Overview of dynNode Class Properties (Node Object)

A **dynNode** class is the primary object that you will be working with. Following is a table providing a quick reference to all exposed properties and methods of the ***dynNode*** class. More details and examples will be provided, including property and method prototypes later on in this document:

Property	Type Decl.	Return Type	Description
Child	Method	dynNode	Get the ***first*** immediate child of this node.
Children	Method	Long	Get the number of immediate child nodes from the *Invoking Node*.
ClearLastError	Method	---	Clear the last error encountered. This is not automatic.
ErrorCode	Prop Get	dynErrorCodes	Get the last error code. Any time you encounter a problem, such as creating a new node and you find that the new object is still defined a ***Nothing***. This error code will inform you of the specific error, and the **ErrorNode** method can be used to obtain a reference to the Node that the error occurred in.
ErrorDescription	Method	String	Returns text description of the node error number.
ErrorMsgBox	Prop Get/Let	Boolean	If set to True, node errors featuring the error number and description are display in a message box. This is very useful during the application development phase, and is highly recommended.
ErrorNode	Prop Get	dynNode	Returns the node where the last error occurred. Nothing if no error.
FindID	Method	dynNode	Get a reference to the node containing the specified ID tag, searching from the root node.
FindIDLocal	Method	dynNode	Get a reference to the Node containing the specified ID tag, searching from the Invoking Node.
FindKey	Method	dynNode	Get a reference to a node matching Key (exact or partial), searching from the root node.
FindKeyLocal	Method	dynNode	Get a reference to a node matching Key (exact or partial), searching from the Invoking Node.
FirstSibling	Method	dynNode	Get the first sibling of this node (sharing this node's parent). This is like invoking **Node.Parent.Child**.
ID	Prop Get	Long	Get the node's unique ID number.
Init	Method	---	*Special* init routine used ONLY for creating *root nodes*.
IsAncestor	Method	Long	Return 1 if the node is the ancestor of a specified node, -1 if it is not, and 0 if there was an error.
IsDescendant	Method	Long	Return 1 if the node is a descendant of a specified node, -1 if it is not, and 0 if there was an error.
IsRoot	Method	Boolean	Returns True if this node is a Root Node
Key	Prop Get/Let	String	Get/Assign unique Key text.
KeyChecks	Prop Get/Let	Boolean	Ignore checks for pre-existence of Keys during **Add** *Nodes Collection* method. This is a very dangerous, though also a very useful property.
KeyExists	Method	Boolean	Return True if the specified key exists in the tree
LastSibling	Method	dynNode	Get the last sibling of this node (sharing this node's parent).
Locked	Prop Get/Let	Boolean	Flag to lock a node from deletion. If set to True, the node cannot be deleted using the Clear or Remove methods exposed by the **Nodes** Collection object. Sibling nodes without this flag set will be deleted, but parent nodes, if they are to be deleted either by the **Nodes.Remove** method or the **Nodes.Clear** methods, will not be deleted if any descendants are locked. All descendants of a Locked node will not be deleted unless you *specifically* specify them, and they are not also tagged as locked.

LockedCount	Method	Long	Get the number of descendant nodes marked **Locked** from this branch, to include this branch.
LockedList	Method	dynNode()	Get an array of Node references to all nodes marked **Locked** from this branch, including this branch.
Marker	Prop Get/Let	Boolean	Get/Assign user-defined Boolean marker. The Node classes do not employ this flag, and it is set aside exclusively for user-use and definition.
MarkerCount	Method	Long	Get the number of descendant nodes marked from this branch.
MarkerList	Method	dynNode()	Get an array of references to all nodes objects marked from this branch.
Move	Method	Boolean	Change the node's parent. Return True if success. If False, check **ErrorCode**. Moving to a new Tree generates a new ID values for all descendant nodes. Moving to a *Nothing* parent converts the node into a new Root-Level Node. Move a Root-Level Node to a branch of another Tree converts the moved Node into a normal Node.
NextSibling	Method	dynNode	Get the next consecutive sibling of this node.
NodeCount	Method	Long	Get a count of all descendant nodes from the current node, inclusive of the current node. This includes all consecutive generations that branch from it.
NodeList	Method	dynNode()	Get a list of all descendant nodes from the specified node, inclusive of the specified node.
Nodes	Prop Get	dynNode()	Returns a reference to a ***dynNodes*** class that in turn potentially contains a list of references to child nodes for the current node.
Parent	Prop Get	dynNode	Get the parent node of the current node.
PreviousSibling	Method	dynNode	Get the previous consecutive sibling of this node.
ResetAllMarkers	Method	---	Set all nodes in the tree to Marked = False.
Root	Method	dynNode	Get the Root-Level Node for this tree.
SortChildren	Method	Boolean	Sort all child nodes (and optionally descendant branches) without setting the **Sorted** flag. Return True if successful. Very useful for one-time sorts.
SortDescending	Prop Get/Let	Boolean	False (Default) = Sort Ascending. True = Descending. This is only applicable when the **Sorted** property is set to True.
Sorted	Prop Get/Let	Boolean	Set/Check sorting on the current node's immediate children. Must be set for each branch.
Tag	Prop Get/Let	Variant	Get/Assign Tag. This is a user-defined field not actually used by the node classes. You can store any type of data in this field.
Text	Prop Get/Let	String	Get/Assign Text. This is the **Node** class's default property. You can access and assign this by specifying **Node.Text** or **Node,** though this latter use because it removes from code self-documentation. Effectively, using this as the default property allows a developer working with this **Node** objects in single-step debug mode to view the Text contents of the active node in code within a *tooltip*.
UnlockAll	Method	---	Set all nodes in the tree to Locked = False.
UserObject	Prop Get/Set	Object	Get or Set a user-defined object into the node
UserVariant	Prop Get/Let	Variant	Get/Assign a user-defined variable or user-defined type (structure). This is just like the Tag property.
Version	Method	String	Get DLL Version (Format: *Major.Minor.Revision*).

dynNode Details (Node Class)

As covered previously, creating a root node requires a simple additional step, which can be eliminated by using the ***NewNodeList()*** function included in the **moddynNode.bas** module. Even if you do not use this function, the step is simply to remember to initialize the new root-level node with a starting Key and Text using the node's ***Init()*** method. Once the root node is created, you can then access its other properties and methods, as listed in the brief overview. Let's now look at those properties and methods.

In all of the examples presented below, it is assumed that the root node object variable is named ***RootNode***, and the "current" node is defined as ***MyNode***.

Child (Node Object)

VB Prototype: `Function Child() As dynNode`

The ***Child*** method returns a Node reference to the first child in a ***Nodes*** *Collection*. The first child is logically the first child referenced in the list of child nodes of the current node. This is equivalent to specifying ***Node.Nodes(1)*** or ***Node.Nodes.Item(1)***, except that if there are no child nodes, an object reference to *Nothing* will be returned, without generating an error.

Syntax

*Node.***Child**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

A reference to the new ***dnyNode*** object. *Nothing* if there are no child nodes.

Child Example:

```
Dim Node As dnyNode
Set Node = MyNode.Child
If Not Node Is Nothing Then Debug.Print Node.Text
```

NOTE: *This Method can generate trappable errors*

Children (Node Object)

VB Prototype: `Function Children() As Long`

The ***Children*** property returns the number of nodes contained in the indicated Node's *Nodes Collection*. This is similar to invoking `MyNode.Nodes.Count`.

Syntax
*Node.***Children**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A long integer containing number of child nodes that share the current node as their parent.

Children Example:
```
Debug.Print CStr(MyNode.Children)
```

ClearLastError (Node Object)

VB Prototype: `Sub ClearLastError()`

The ***ClearLastError*** method clears any flagged errors that were encountered. This command is useful if you are about to execute a Node command. This is most important when you believe that you are not receiving the results that you believe that you should be receiving. During initial program development, it may be wise to set the ***ErrMsgBox*** property to True, as well. This way, when there are any Node error occurs, you will be notified with a message box report.

Syntax
*Node.***ClearLastError**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
No return value.

ClearLastError Example:

```
Dim Node as dynNode
MyNode.ClearLastError
Set Node = MyNode.Nodes.Add(, dynNodeChild,"New Child","Herman")
If MyNode.ErrorCode <> dynNodeErrorSuccess Then    'if the error code is not 0
  MsgBox MyNode.ErrorDescription(MyNode.ErrorCode) 'Handle error here
  Debug.Assert False                               'Let the developer add debugging
code
End If
```

NOTE: *This Method can generate trappable errors*

ErrorCode (Node Object)

VB Prototype: `Property Get ErrorCode() As dynErrorCodes`

The ***ErrorCode*** property lets you know if an error occurred. This property returns a value of type ***dynNodeErrorCodes***. If there was no error, the return code is ***dynNodeErrSuccess (0)***.

Syntax
*Node.***ErrorCode**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
No return value.

The error code values for ***dynNodeErrorCodes*** are defined as follows:

Enumerated Name	Value	Description
dynNodeErrSuccess	0	No error. Everything is OK
dynNodeErrNotInitialized	1	The *Root-Level Node* is not yet initialized by the ***Init()*** method.
dynNodeErrAlreadyInitialized	2	The *Root-Level Node* is already initialized. The user is attempting to invoke the ***Init()*** method on it again. *This is not a fatal error*, but any specified Key and Text that was included with the ***Init()*** invocation will not be set. You can alter these by changing the node's ***Key*** and ***Text*** properties.
dynNodeErrNodeIsNothing	3	The indexed or a specified node does not exist.
dynNodeErrParentAreSame	4	The node and the specified node share the same parent. This will normally occure with a Move() method. *This is not a fatal error*, and nothing is done.
dynNodeErrNodesAreTheSame	5	This node is the same as the specified node. *This is not a fatal error*, and nothing is done.
dynNodeErrAncestorDescendantConflict	6	There was an ancestor/descendant conflict between two specified nodes in a ***Move()*** method invocation. Nothing was done.
dynNodeErrIndexOutOfRange	7	The specified index is out of range in a *Nodes Collection*. The index was either less than 1, or greater than the ***Nodes.Count*** value. Nothing was done.
dynNodeErrRelationshipRequiresIndex	8	The specified *Relationship* during an ***Add()*** method invocation requires an Index or Key be supplied. Nothing was done.
dynNodeErrKeyAlreadyExists	9	An attempt was made to either change a node' *Key* value, or add a new Node with a *Key* value that already exists in the Tree. Nothing was done.
dynNodeErrUserKeyIsNumeric	10	A user-supplied key cannot be numeric. Numeric keys defined by the Add() method when the user does not supply a key. In this case it will be the value of the Node's ID property. By preventing the user from setting numeric Keys (or Keys that begin with a digit), the structure can be assured of not duplicating Key during creation. Nothing was done.

dynNodeErrKeyInvalid	11	The user-supplied key is not an existing string or index. Nothing was done.
dynNodeErrKeyParameterBlank	12	The user did not supplied a key or index. Nothing was done.
dynNodeErrIdOutOfRange	13	The specified ID number is out of range. The user invoked the FindID() method with a ID value of less than 1, or the Node assigned that ID value no longer exists. Nothing was done.
dynNodeErrRootCanHaveNoSiblings	14	A sibling function was attempted on the *Root-Level Node*. A Root-Level Node cannot have siblings; only descendants. Nothing was done.
dynNodeErrInvalidIndexValue	15	The user supplied an Index that was either non-integer or non-long integer.

For an example, see ***ClearLastError***.

NOTE: *This Method can generate trappable errors*

ErrorDescription (Node Object)

VB Prototype:

```
Function ErrorDescription(ErrorNumber As dynErrorCodes) As String
```

The ***ErrorDescription*** method, provided a value of type ***dynNodeErrorCodes***, returns a text description of the error encountered.

Syntax

*Node.***ErrorDescription(*ErrorCode*)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

ErrorCode is a *dynNodeErrorCode* value for which to return the string description.

Returns

A string with the description assigned to the error code.

For an example, see ***ClearLastError***. See ***ErrorCode*** for the list of *Error Code* values.

NOTE: *This Method can generate trappable errors*

ErrorMsgBox (Node Object)

VB Prototype:
```
Property Get ErrorMsgBox() As Boolean
Property Let ErrorMsgBox(ShowErrorsInMsgBox As Boolean)
```

The ***ErrorMsgBox*** property will tell the Nodes to display message boxes when any node errors are encountered, or not. Though it is not recommended to have this property enabled for 'release' products, it is extremely helpful during application development.

Syntax
*Node.***ErrorMessageBox**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
True if ***ErrorMsgBox*** is enabled, otherwise **False**.

ErrorMsgBox Example:
```
RootNode.ErrorMsgBox = True                    'set error reporting to message box
Dim NewNode As dynNode
Set NewNode = RootNode.Add(, dynNodeLast,"New Root Sibling","BadIdea") 'Gen error
```

ErrorNode (Node Object)

VB Prototype: `Property Get ErrorNode() As dynNode`

The ***ErrorNode*** property will return a reference to the node that generated the last error. Be sure to first check the ***ErrorCode*** property for a value other than *dynNodeErrSuccess* (or 0) before getting this node reference.

Syntax
*Node.***ErrorMessageBox**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A Node reference to the node that generated the last error when the ***ErrorCode*** property is not set to *dynNodeErrSuccess*.

ErrorNode Example:

```
If RootNode.ErrorCode <> 0 Then
  Dim ErrNode As dynNode
  Set ErrNode = RootNode.ErrorNode  'get error node
  Debug.Print "Node " & ErrNode.Text & " encountered an error:" & vbCrLf & _
              RootNode.ErrorDescription(RootNode.ErrorCode)
End If
```

NOTE: *This Method can generate trappable errors*

FindID (Node Object)

VB Prototype: `Function FindID(ByVal ID As Long) As dynNode`

The ***FindID*** method will return a reference to the node with the specified ID number. Searching through the entire tree.

Syntax
Node.**FindID(IDnumber)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

IDNumber is a Long Integer value which represents the unique ID number for the sought node.

Returns
A Node reference to the node that contains the specified ID number. *Nothing* will be returned if that ID does not exist within the tree.

FindID Example:

```
Dim tmpNode As dynNode
Set tmpNode = RootNode.FindID(1234) 'Search for node with ID of 1234
If Not tmpNode Is Nothing Then      'If the node was found...
  Debug.Print tmpNode.Text          'display its Text property
End If
```

NOTE: *This Method can generate trappable errors*

FindIDLocal (Node Object)

VB Prototype: `Function FindIDLocal(ByVal ID As Long) As dynNode`

The ***FindID*** method will return a reference to the node with the specified ID number, searching from the Invoking Node downward through its branch.

Syntax
*Node.***FindIDLocal(IDnumber)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

IDNumber is a Long Integer value which represents the unique ID number for the sought node.

Returns
A Node reference to the node that contains the specified ID number. *Nothing* will be returned if that ID does not exist within the branch of the tree from the invoking Node.

FindID Example:

```
Dim tmpNode As dynNode
Set tmpNode = MyNode.FindIDLocal(1234)   'Search for node with ID of 1234 from current
node
If Not tmpNode Is Nothing Then           'If the node was found...
  Debug.Print tmpNode.Text               'display its Text property
End If
```

NOTE: *This Method can generate trappable errors*

FindKey (Node Object)

VB Prototype: `Function FindKey(Key As String) As dynNode`

The ***FindKey*** method will return a reference to the node with the specified Key. Searching through the entire tree.

Syntax
*Node.***FindKey(Key)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Key is a text string which represents the unique Key for the sought node. This string is alphanumeric, and cannot begin with a numeric digit.

Returns
A Node reference to the node that contains the specified Key. *Nothing* will be returned if that Key does not exist within the tree.

FindKey Example:

```
Dim tmpNode As dynNode
Set tmpNode = RootNode.FindKey("C:\Level1\Level1-2\Test")  'Search for node with a
'dir' key
If Not tmpNode Is Nothing Then                             'If the node was found...
  Debug.Print tmpNode.Text                                 'display its Text property
End If
```

NOTE: *This Method can generate trappable errors*

FindKeyLocal (Node Object)

VB Prototype: `Function FindKeyLocal(Key As String) As dynNode`

The ***FindKeyLocal*** method will return a reference to the node with the specified Key, searching from the Invoking Node downward through its branch.

Syntax

*Node.***FindKeyLocal(IDnumber)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Key is a text string which represents the unique Key for the sought node. This string is alphanumeric, and cannot begin with a numeric digit.

Returns

A Node reference to the node that contains the specified Key. *Nothing* will be returned if that Key does not exist within the branch of the tree from the invoking Node.

NOTE: *This Method can generate trappable errors*

FirstSibling (Node Object)

VB Prototype: `Function FirstSibling() As dynNode`

The ***FirstSibling*** method will return a reference to the node that is the first sibling of the list of nodes which share the Invoking Node's parent Node.

Syntax

Node.**FirstSibling()**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

A Node reference to the node that is the first child in the list of child nodes which share the Invoking Node's parent node.

NOTE: *This Method can generate trappable errors*

ID (Node Object)

VB Prototype: `Function ID() As Long`

The ***ID*** property returns a Long Integer of a value that is automatically generated for each node. This uniquely identifies each note. This is also the value used to generate the Key value if you create a node without supplying a Key. Hence each node contains two unique values, the Key and the ID. The advantage for a developer is that by providing a Key, the key can consist of text that makes sense to the developer. This property cannot be changed by the user.

Syntax

Node.**ID**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Example:

```
Debug.Print "This node's ID number is " & CStr(MyNode.ID)
```

Returns

The unique ID number of the Invoking Node.

Init (Node Object)

VB Prototype:

```
Public Sub Init(Optional Key As String, Optional Text As String)
```

The ***Init*** method is used for the initialization of *Root-Level Nodes*. Unlike other nodes of the tree, a *Root-Level Node* does not have a parent node and has no siblings. All branches of the tree converge on it at their "root" level. Because it does not have a parent, it cannot be created using the *Add* method from a yet non-existent ***Nodes Collection***. Because it needs to have its Key and Text properties set, the ***Init()*** method is available do perform this task. Once this method has been invoked, invoking it again, or invoked on a node that is not a *Root-Level Node*, it will generate the internal error *dynNodeErrAlreadyInitialized*, but this is not fatal and will not actually perform any operations, or change the Key or Text values previously assigned to the node.

Syntax

*Node.***Init***([Key], [Text])*

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Key is a text string which represents the unique Key for the new node. This string is alphanumeric, and cannot begin with a numeric digit (Keys with numeric values are generated automatically when the program does not provide a key).

Text is a text string which represents the "public" text for the new node. This will be the text displayed in a *tooltip* during debugging when a node reference is hovered over. It can also represent a displayable folder or file name in a directory.

Example:

```
Dim cNd As New dynNode  'define new root-level node
cNd.Init Key, Text      'initialize it (do this on your root-level node before anything
else)
```

IsAncestor (Node Object)

VB Prototype: `Function IsAncestor(OfNode As dynNode) As Long`

The ***IsAncestor*** method will return a Boolean value indicating if the Invoking Node is an ancestor of the specified Node reference.

Syntax

Node.**IsAncestor(OfNode)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

OfNode is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

True if the Invoking Node is an ancestor of the specified Node, otherwise *False*.

NOTE: *This Method can generate trappable errors*

IsDescendant (Node Object)

VB Prototype: `Function IsDescendant(OfNode As dynNode) As Long`

The ***IsDescendant*** method will return a Boolean value indicating if the Invoking Node is a descendant of the specified Node reference.

Syntax

Node.**IsDescendant(OfNode)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

OfNode is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

True if the Invoking Node is a descendant of the specified Node, otherwise *False*.

NOTE: *This Method can generate trappable errors*

IsRoot (Node Object)

VB Prototype: `Function IsRoot() As Boolean`

The ***IsRoot*** method will return a Boolean value indicating if the Invoking Node is the *Root-Level Node* of the tree containing the invoking Node.

Syntax
Node.**IsRoot**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
True if the Invoking Node is the Root-Level Node of the tree, otherwise *False*.

NOTE: *This Method can generate trappable errors*

Key (Node Object)

VB Prototype:
```
Property Get Key() As String
Property Let Key(KeyText As String)
```

The ***Key*** property is a unique text string that will be used to uniquely identify the node. If the user does not specify a Key when they created the node, then the class will automatically use a text version of its unique ID value. A user-defined Key must begin with an Alphabetic character, and cannot be numeric (the danger of duplicating other ID values would otherwise exist). The user can change this property, but the new Key value must still be unique. Check the ***ErrorCode*** property after setting this if the key does not seem to "take".

Syntax
*Node.***Key**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Example:
```
Debug.Print MyNode.Key                'display current key
MyNode.Key = "Inventory Schedules"    'Change the key for this node
Debug.Print MyNode.Key                'check it for taking
```

KeyChecks (Node Object)

VB Prototype:
```
Property Get KeyChecks() As Boolean
Property Let KeyChecks(Enabled As Boolean)
```

The ***KeyChecks*** property enables or enabled the *Key* checking flag during the ***Add*** method of the ***Nodes*** *Collection*. The *default* and *recommended* state is *True*. Though this is a potentially dangerous command, it is also very safe and powerful if you can *guarantee* that building your node tree will not duplicate pre-existing Key values of other nodes already in the tree. Normally, this property should be left alone in its default True state, as it is important that all node indeed have a unique *Key*. However, you can cut build times by several hundred percent if you know that the Keys that you will be generating will not conflict. Setting or resetting this property from any valid Node will set the flag for *all* nodes in the tree it shares. For example, enumerating a unique list of drive folders that could exceed 10,000 folders will take half the time it would normally (safely) take. Enumerating a list of words into a dictionary of 5000 unique words will take less than one second with the check off, as opposed to 20 seconds with the check on.

If you do build a list with this property turned off (set to *False*), and you plan on adding items later, *please* be sure, after initially building the list, to turn this property back on (Set to its *default* state of *True*). It is highly recommended that you leave this property alone. It is simply made available for those who can *guarantee* that they *Key* values will never conflict while they are using it with this property disabled.

Syntax
Node.**KeyChecks**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

KeyChecks Example:
```
Set RootNode = New dynNode  'create new root-level node
RootNode.Init Path, Path    'initialize it
RootNode.KeyChecks = False  'we know we will not run into duplicates...
'.................................
'...Tree-building code goes here...
'.................................
RootNode.KeyChecks = True   'Turn option back to default for safety!!!
```

KeyExists (Node Object)

VB Prototype: `Function KeyExists(Key As String) As Boolean`

The ***KeyExists*** method will return a Boolean value indicating if the specified unique Key is used by a node within the tree containing the *Invoking Node*.

Syntax

*Node.***KeyExists(Key)**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Key is a text string which represents the unique Key for the node to check for.

Returns

True if the indicated Key is assigned to a node within the tree containing the *Invoking Node*, otherwise *False*.

NOTE: *This Method can generate trappable errors*

LastSibling (Node Object)

VB Prototype: `Function LastSibling() As dynNode`

The ***LastSibling*** method will return a reference to the node that is the last sibling of the list of nodes which share the Invoking Node's parent Node (technically, this means the node whose ***Index*** property is the highest in the child node list that the *Invoking Node* is a member of).

Syntax
*Node.***LastSibling**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A Node reference to the node that is the last child in the list of child nodes which share the Invoking Node's parent node.

NOTE: *This Method can generate trappable errors*

Locked (Node Object)

VB Prototype:
```
Property Get Locked() As Boolean
Property Let Locked(ByVal value As Boolean)
```

The ***Locked*** property allows you to lock, or tell if the *Invoking Node* is locked. When a node is locked, it and all of its descendants will be protected from removal when the locked node is selected for removal, *except* when the root node is removed by setting it to *Nothing*. Descendant nodes can still be removed individually, as long as they are not also locked. If an ancestor node of a locked node is being removed, it will not be removed as long as any of its descendants are locked, though other of its children will be removed as long as they, or their descendant are not also locked. Hence, if a node has 10 children, and one of its children has another 10 children, but only one of these grandchildren is locked, then the other nine unlocked grandchildren will be removed, the locked grandchild will remain, and of the 10 children of the node being removed, the other 9 will be removed. In the end the node *to be removed* will ***not*** in this case be removed; it will only have one child node remaining that will not be removed, because one of that node's children is locked, though the other children (grandchildren of the target node for removal) are removed. This would leave just three nodes total remaining (not counting the descendants of the locked grandchild). The reason direct-line ancestor nodes are not also removed is there would subsequently be no way of accessing the locked node (otherwise it would be totally pointless to have locked nodes).

Syntax
*Node.***Locked**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A Boolean value that is *True* if the node is locked, and *False* if it is not locked.

NOTE: *This Property can generate trappable errors*

LockedCount (Node Object)

VB Prototype: `Function LockedCount() As Long`

The ***LockedCount*** method returns the number of descendant nodes, to include the *Invoking Node*, that are locked.

Syntax
*Node.***LockedCount**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A Long Integer value indicating the number of nodes from the *Invoking Node* that are locked.

NOTE: *This Method can generate trappable errors*

LockedList (Node Object)

VB Prototype: `Function LockedList() As dynNode()`

The ***LockedList*** method returns an array of type ***dynNode***, containing references to all descendant nodes, to include the *Invoking Node*, that are locked.

Syntax
*Node.***LockedList**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
An array of ***dynNode*** references. If no nodes are locked, then the returned array will not be dimensioned. It would therefore be wise to first obtain the ***LockedCount*** property value to determine if you should attempt to obtain a list.

LockedList Example:

```
Dim Count As Long
Count = MyNode.LockedCount     'get number of locked nodes from the current node
If Count <> 0 Then             'if any nodes are locked
  Dim TmpList() As dynNode     'list to receive the list
  Dim Index As Long            'Looping Index
  TmpList = MyNode.LockedList 'Get the list of locked nodes
  For Index = 0 To Count - 1   'Display each locked node's Key property
    Debug.Print TmpList(Index).Key
  Next Index
End If
```

NOTE: *This Method can generate trappable errors*

Marker (Node Object)

VB Prototype:
```
Property Get Marker() As Boolean
Property Let Marker(ByVal value As Boolean)
```

The ***Marker*** property is a user-definable feature that allows you to set a flag, or tell if the *Invoking Node* is marked. The *Marker* flag is not used by any other featured of ***DynamicNodes***. The user can employ it however they wish. For example, the user may wish, if creating a tree, to use it to easily distinguish between nodes that are branches, and those that are leaves. Or, they may wish to mark certain nodes for later special processing.

Syntax
Node.**Marker**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A Boolean value that is *True* if the node is marked, and *False* if it is not marked.

NOTE: *This Property can generate trappable errors*

MarkerCount (Node Object)

VB Prototype: `Function MarkerCount() As Long`

The ***MarkerCount*** method returns the number of descendant nodes, to include the *Invoking Node*, that are marked.

Syntax
Node.**MarkerCount**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A Long Integer value indicating the number of nodes from the *Invoking Node* that are marked. Hence, the returned value will always be at least one.

NOTE: *This Method can generate trappable errors*

MarkerList (Node Object)

VB Prototype: `Function MarkedList() As dynNode()`

The ***MarkerList*** method returns an array of type ***dynNode***, containing references to all descendant nodes, to include the *Invoking Node*, that are marked.

Syntax
*Node.***MarkerList**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
An array of ***dynNode*** references. Because the *Invoking Node* will be included in this list, the returned array will always have a dimension that is at least 0 (1 item).

NOTE: *This Method can generate trappable errors*

Move (Node Object)

VB Prototype: `Function Move(NewParent As dynNode) As Boolean`

The ***Move*** method allows you to move a node from one location in a tree to another, to include moving it to another ***DynamicNodes*** list. All descendant nodes from the moved node will also be moved. An Boolean False return code will be sent if the user attempts to move a node to one of its own descendant positions. Note that you can create a new tree by simply setting the ***NewParent*** property to *Nothing*, as in `Call MyNode.Move(Nothing)`.

Syntax

Node.**MarkerList**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

NewParent is an object expression that resolves to a ***dynNode*** object that points to a valid reference. If ***NewParent*** is *Nothing*, then the node will be converted to a new Root-Level Node for a separate tree with a new ID value set to 1. The ID values of any of its descendant nodes will also be recomputed. If the Invoking Node was a *Root-Level Node*, and ***NewParent*** is a node in another tree, the *Invoking Node* will be converted to a normal node, and it, along with any descendants, will have their ID values recomputed in order not to interfere with the ID values of the new parent tree.

Returns

A Boolean value that is *True* if the node is moved, and *False* if it is not moved.

NextSibling (Node Object)

VB Prototype: `Function NextSibling() As dynNode`

The ***NextSibling*** method will return a reference to the node that is the sibling immediately following the *Invoking Node* in the list of nodes which share the *Invoking Node's* parent Node (technically, this means the node whose ***Index*** property is one more than the ***Index*** property of the *Invoking Node*).

Syntax

*Node.***NextSibling()**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

A Node reference to the node that is the next sequential sibling of the *Invoking Node* in the list of child nodes which share the *Invoking Node's* parent node. If there is no node listed after the *Invoking Node*, then *Nothing* will be returned.

NOTE: *This Method can generate trappable errors*

NodeCount (Node Object)

VB Prototype: `Function NodeCount() As Long`

The ***NodeCount*** method returns the number of descendant nodes, to include the *Invoking Node*.

Syntax
Node.**MarkerCount**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A Long Integer value indicating the number of descendant nodes from the *Invoking Node*, to include the *Invoking Node*. Hence, the returned value will always be at least one.

NOTE: *This Method can generate trappable errors*

NodeList (Node Object)

VB Prototype: `Function NodeList() As dynNode()`

The ***NodeList*** method returns an array of type ***dynNode***, containing references to all descendant nodes, to include the *Invoking Node*. The Array bounds are 0 to the *UBound* value of the array variable.

Syntax

*Node.***NodeList**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

An array of ***dynNode*** references. Because the *Invoking Node* will be included in this list, the returned array will always have a dimension that is at least 0 (1 item).

NOTE: *This Method can generate trappable errors*

Nodes (Node Object)

VB Prototype: `Property Get Nodes() As dynNodes`

The ***Nodes*** property returns a reference to the *Invoking Node's* ***Collection Object***. Normally this reference is made for reasons of further invoking a method or property of the ***Nodes*** object. If you will be performing a lot of operations on this object in a routine, note that you can speed access by defining a nodes reference variable of type ***dynNodes*** and assigning it (use the ***Set*** keyword but without the ***New*** keyword – remember that we are not needing to instantiate a new ***dynNodes*** object, but rather only obtaining a reference to one that already exists).

Syntax

Node.Nodes

Returns

An reference to a ***dynNodes*** object.

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Nodes is an object expression that resolves to a ***dynNodes*** object.

See the ***Nodes Object*** section listed previously to examine the properties and methods that it provides.

Parent (Node Object)

VB Prototype: `Property Get Parent() As dynNode`

The ***Parent*** property will return a reference to the node that is the parent node of the *Invoking Node* (the node that contains the *Invoking Node* as a child).

Syntax

Node.**Parent**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

A Node reference to the node that is the parent of the *Invoking Node*. A *Root-Level Node* is the only node that will return a Nothing object reference.

PreviousSibling (Node Object)

VB Prototype: `Function PreviousSibling() As dynNode`

The ***PreviousSibling*** method will return a reference to the node that is the sibling immediately previous to the *Invoking Node* in the list of nodes which share the *Invoking Node's* parent Node (technically, this means the node whose ***Index*** property is one less than the ***Index*** property of the *Invoking Node*).

Syntax

*Node.***NextSibling()**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

A Node reference to the node that is the next child in the list of child nodes which share the *Invoking Node's* parent node. If there is no previous node to the *Invoking Node* in the list, then *Nothing* will be returned.

NOTE: *This Method can generate trappable errors*

ResetAllMarkers (Node Object)

VB Prototype: `Sub ResetAlllMarkers()`

The ***ResetAllMarkers*** method resets the user-definable marker flag of the *Invoking Node* and all marked nodes of any of its descendants. If you use the ***Marked*** property to flag certain nodes for special processing, this is a quick way to reset those user-defined markers from a branch or tree.

Syntax

*Node.***ResetAllMarkers()**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

No return value.

NOTE: *This Method can generate trappable errors*

Root (Node Object)

VB Prototype: `Function Root() As dynNode`

The ***Root*** method will return a reference to the *Root-Level Node* that is primary ancestor of all nodes in the tree containing the *Invoking Node*. Though normally the scope of your Root-Level Node reference should be broad enough to make it available to your program code, this method is available for situations where that situation might not be so.

Syntax

*Node.***Root()**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns

A Node reference to the Root-Level Node of the tree containing the Invoking Node.

NOTE: *This Method can generate trappable errors*

SortChildren (Node Object)

VB Prototype:

```
Function SortChildren(Optional SortAllGenerations As Boolean = False, _
                      Optional ReverseOrder As Boolean = False) As Boolean
```

The ***SortChildren*** method will force an alphabetical sort of the child nodes of the *Invoking Node*. The default sort method is ascending. This method is not pendant on the ***Sorted*** or ***SortDescending*** properties. This method is best used when you require a one-time sort of an entire tree, though your application may not require setting the ***Sorted*** property, for example.

Syntax

*Node.***SortChildren([SortAllGenerations] [,ReverseOrder])**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

ReverseOrder is a Boolean value that, when set to True, will force a sort in reverse alphabetical order. The default value of the parameter is False, which invokes an ascending alphabetical sort.

Returns

True if sorting is successfully completed. If False, an error was generated.

NOTE: *This Method can generate trappable errors*

SortDescending (Node Object)

VB Prototype:
```
Property Get SortDescending() As Boolean
Property Let SortDescending (ByVal SetSort As Boolean)
```

The ***SortDescending*** property forces sorting the child nodes of the *Invoking Node* in ascending or descending alphabetical order if the Sorted property is assigned a value of True. If ***SortDescending*** is assigned a value of True, then the alphabetical sort will be in descending order. If assigned a value of False (default), then sorting will be in ascending alphabetical order. If the ***Sorted*** property is already set to True, and the ***SortDescending*** property is set to a state opposite of its previous state, then a sort of the *Invoking Node's* immediate child nodes will be invoked.

Syntax
*Node.***SortDescending**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A variable of type Boolean. True indicates that child nodes of the *Invoking Node* will be sorted in descending alphabetical order if the ***Sorted*** property is also set to True.

See also: the ***Sorted*** property and the ***SortChildren*** method.

Sorted (Node Object)

VB Prototype:
```
Property Get Sorted() As Boolean
Property Let Sorted(ByVal SetSort As Boolean)
```

The ***Sorted*** property forces sorting the child nodes of the *Invoking Node* in alphabetical order if assigned a value of True. If assigned a value of False (default), then no sorting will take place. While this property is set to True, the child node list of the *Invoking Node* will automatically resort its child list when a new child node is added.

Syntax
*Node.***Sorted**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A variable of type Boolean. True indicates that child nodes of the *Invoking Node* will be sorted in alphabetical order.

See also: the ***SortDescending*** property and the ***SortChildren*** method.

Tag (Node Object)

VB Prototype:
```
Property Get Tag() As String
Property Let Tag(TagText As String)
```

The ***Tag*** property allows you to set or retrieve a user-defined string. This property is not used by ***DynamicNodes***, and can be employed for whatever the user wishes.

Syntax
*Node.***Tag**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A variable of type String. It returns *vbNullstring* (blank) if no text has been assigned to the *Tag* property.

Text (Node Object) (DEFAULT Property)

VB Prototype:
```
Property Get Text() As String
Property Let Text(TextText As String)
```

The ***Text*** property allows you to set or retrieve a user-defined string. This property is not altered by ***DynamicNodes***, and can be used for whatever the user wishes, though ***DynamicNodes*** does employ it during program development as the *Tool-Tip* contents of the node when the developers points to a node reference in the program code.

Syntax
*Node.***Text**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A variable of type String. It returns *vbNullstring* (blank) if no text has been assigned to the *Text* property.

UnlockAll (Node Object)

VB Prototype: `Sub UnlockAll()`

The ***UnlockAll*** method will reset the ***Locked*** property on all descendant nodes, to include the *Invoking Node*, to *False*. This task is also automatically performed when you set a Node to *Nothing*.

Syntax

*Node.***UnlockAll**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

NOTE: *This Method can generate trappable errors*

UserObject (Node Object)

VB Prototype:
```
Property Get UserObject() As Boolean
Property Set UserObject(UsrObject As Object)
```

The ***UserObject*** property allows you to store and acquire an object reference in the *Invoking Node*. This is a user-definable property, and is not used by ***DynamicNodes***. This property is set to *Nothing* to release the reference when the node is released. Remember than when assigning ***UserOject***, or assigning its property to an object reference variable, to use the ***Set*** keyword.

Be careful to not present a situation where you may present yourself with reference issues where you may assign an object to ***UserObject***, later release the object, but not the reference in ***UserObject***. COM has a habit of resurrecting released objects if a reference to the released object still exists, and the reference is used to access the object. This is not an issue created by ***DynamicNodes***, but a condition that is prevalent (quite frankly, a pain in the butt) in the nature of COM objects. With a good understanding of the COM architecture, objects, and references, you should have little trouble in this department (an excellent reference in this regard is the ***Microsoft Visual Basic 6.0 Programmer's Guide*** from *Microsoft Press*, ISBN 1-57231-873-2).

Syntax
*Node.***UserObject**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A reference to an assigned object. It returns *Nothing* if no object has been assigned to the ***UserObject*** property.

UserVar (Node Object)

VB Prototype:
```
Property Get UserVar() As Variant
Property Set UserVar(UsrObject As Object)
Property Let UserVar(UsrVariant As Variant)
```

The ***UserVar*** property allows you to store and acquire an variant or any other type variable in the *Invoking Node*. This is a user-definable property, and is not used by ***DynamicNodes***. Normally, you should use the ***UserObject*** property for storing objects, but the support of adding objects to the ***UserVar*** storage location allows you to easily store more than one object in a node.

Syntax
*Node.***UserVariant**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Returns
A variable of type Variant. It returns *Null* if no variable has been assigned to the *UserVar* property.

Version (Node Object)

VB Prototype: `Function Version() As String`

The Version property returns the DLL version number in the format "Major.Minor.Revision". Hence, if Major was 1, Minior was 0, and Revision was 1, the Version property would return "1.0.1".

Syntax

*Node.***Version()**

Node is an object expression that resolves to a ***dynNode*** object that points to a valid reference.

Example:

```
Debug.Print "The DLL Version number is: " & CStr(MyNode.Version)
```

Handling Error Conditions

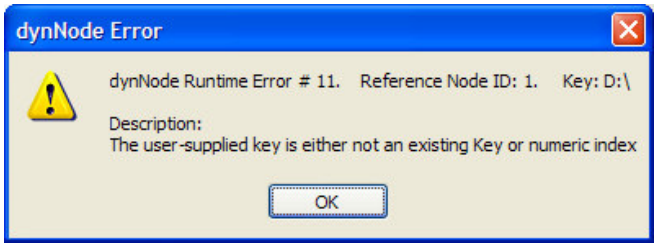
Inevitably, an error in programming logic will cause a Node-based operation to fail. If you examine the ***ErrorCode()*** property for the ***dynNode*** Class in the previous section, you will see the errors that are generated. Because you are not using a public interface such as a *TreeView*, you do not have a regular method of trapping errors that will warn you with a mechanism similar to a TreeView-level error event handling mechanism. However, you have 3 options available to you.

Error Handling Method 1: MessageBox Alerts

The First and easiest method is to handle trapping errors is by setting the ***ErrorMsgBox*** property to *True*. This will display a message dialog box with an OK button that shows you the ***DynNode*** error number, the unique node ***ID*** number of the node that generated the error, the ***Key*** for this node, and a general description of the error. Although this *Quick 'n' Dirty* technique is not a recommended practice for professional-level code that is being shipped commercially, it is perfect during the in-house application development phase of such projects. Hobbyists may find that this error trapping mechanism is easier that the other two methods mentioned below, and much less of a bother, especially for programs they are just playing around with and have no commercial ambitions for them.

From any valid ***dynNode*** reference, such as your stored *Root-Level Node* reference variable, you can simply do the following, assuming that ***RootNode*** is the name that you declared for this node:

```
RootNode.ErrorMsgBox = True
```



Now, in the rare event that an error is generated, a message box will pop up similar to the one on the right.

As you can see from the description, we may have supplied a Key or Index that is not defined in the tree.

Error Handling Method 2: Error Event Support

DynamicNodes supports *Event* processing on errors by you declaring a **Node** reference variable with the ***WithEvents*** keyword. But as would be expected, these event triggers only work on a global or file-local level in a *Class* or *Form*. If you are willing to declare your Node reference variables in such a manner, which can be surprisingly few, then you will find an event named ***dynNodeError*** added to your repertoire of program features associated which each declared Node reference variable.

Event Handler Example:

The following code snippet defines the all-important reference to your *Root-Level Node* (which the following methods assume to be initialized when they are triggered by an error event), and two "scratchpad" node reference variables, along with their simple message box event handlers:

```
Public WithEvents RootNode As dynNode         'define new root node (instantiate
elsewhere)
Public WithEvents dNode1 As dynNode           'define a general local node reference
Public WithEvents dNode2 As dynNode           'define a general local node reference

Private Sub RootNode_dynNodeError(ErrorCode As dynErrorCodes, NodeID As Long)
  MsgBox "Node Error code: " & CStr(ErrorCode) & ", " & _
         "Node ID = " & CStr(NodeID) & ", " & _
         "Name = " & RootNode.FindID(NodeID).Text, , "Node Error"
End Sub

Private Sub dNode1_dynNodeError(ErrorCode As dynErrorCodes, NodeID As Long)
  MsgBox "Node Error code: " & CStr(ErrorCode) & ", " & _
         "Node ID = " & CStr(NodeID) & ", " & _
         "Name = " & dNode1.FindID(NodeID).Text & vbCrLf & _
         "Description: " & dNode1.ErrorDescription(ErrorCode), , "Node Error"
End Sub

Private Sub dNode2_dynNodeError(ErrorCode As dynErrorCodes, NodeID As Long)
  MsgBox "Node Error code: " & CStr(ErrorCode) & ", " & _
         "Node ID = " & CStr(NodeID) & ", " & _
         "Name = " & dNode1.FindID(NodeID).Text & vbCrLf & _
         "Description: " & dNode2.ErrorDescription(ErrorCode), , "Node Error"
End Sub
```

Note that the two parameters for the event are ***ErrorCode*** and ***NodeID***. A reference to the actual Node that generated the error can be found by invoking the ***FindID*** method from any valid node (or by invoking the ***ErrorNode*** method, mentioned soon), as well as a human-readable error description. Note that because these additional items (among others, such as *Key*) can be referenced from any valid node reference, and that the *Root-Level Node* must always exist when other descendant nodes exist, you can simplify your code by in turn invoking a guaranteed-to-exist Node event handler, thus:

```
Private Sub dNode1_dynNodeError(ErrorCode As dynErrorCodes, NodeID As Long)
  RootNode_dynNodeError ErrorCode, NodeID
End Sub

Private Sub dNode2_dynNodeError(ErrorCode As dynErrorCodes, NodeID As Long)
```

```
  RootNode_dynNodeError ErrorCode, NodeID
End Sub
```

Error Handling Method 3: Programmer Error Trapping

A more labor-intensive method is to check for errors being generated in your code as you go. Although this seems like a lot of trouble to the general VB programmer, I think that VB Developers (*Professional-Level Programmers*) will find such things routine, preferring that over field bug reports. This method involves checking the ***ErrorCode*** property of a valid Node reference (the *Root-Level Node* is perfect for this). If the ***ErrorCode*** value it is non-zero (not equal to *dynNodeErrSuccess*), then an error has occurred. The ***ErrorNode*** property will be then set to the node where this error occurred. You can invoke the ***ErrorDescription*** method to obtain a text description of the error. Before invoking you Node-based method or property, or before attempting to trap an error on a node operation you are about to perform, be sure to first invoke the ***ClearLastError*** method so that you will not inadvertently assume that the error you just caught occurred after the current node operation.

Example:

```
  With RootNode
    .ClearLastError                              'clean error trap
    Set Nd = .Nodes("Jurassic Period")           'attemp to grab a node
    '
    ' if we did not get what we should have got, trown debug info to the
    ' IDE's immediate window...
    '
    If Nd Is Nothing Then
      If .ErrorCode <> dynNodeErrSuccess Then    'if there was an error (non-zero)...
        Debug.Print "Error: " & CStr(.ErrorCode) & _
                    ",  Node ID: " & CStr(.ErrorNode.ID) & _
                    ",  Node Key: " & .ErrorNode.Key & vbCrLf & _
                    "Description: " & .ErrorDescription(.ErrorCode)
      End If
    End If
  End With
```

The following ***Nodes*** *Collection* properties and methods can generate trappable errors:

Add
InitForNext
Item
Remove.

The following ***Node*** *properties* can generate trappable errors:

Key
Sorted
SortDescending.

The following ***Node*** *methods* can generate trappable errors:

Child	***IsAncestor***	***Move***
ClearLastError	***IsDescendant***	***NextSibling***
ErrorCode	***IsRoot***	***NodeCount***
ErrorMsgBox	***KeyChecks***	***NodeList***
ErrorNode	***KeyExists***	***PreviousSibling***
FindID	***LastSibling***	***ResetAllMarkers***
FindIDLocal	***LockedCount***	***Root***
FindKey	***LockedList***	***SortChildren***
FindKeyLocal	***MarkerCount***	***UnlockAll***
FirstSibling	***MarkerList***	

Recursion Example

Some people tend to look at the idea of recursive programming with fear and apprehension. However, with hierarchical lists such as ***DynamicNodes***, *TreeViews*, and disc directories, recursions is a *perfect* searching method for *quickly* finding information. Recursion into a hierarchical lists, such a node system or a directory structure is often referred to as "drilling down" through the list, because it will call itself for each member node to each of its child nodes, which will in turn do the same, for as many generations of nodes that exist. But once a match is found, it very quickly unwinds itself by returning from the embedded calls in very fast order, effectively discontinuing the search.

For example, the **Node**'s ***FindID*** and ***FindKey*** methods, among others, use a simple but extremely fast recursion method to find a particular *ID* or *Key*. Suppose you were using the user-defined ***Tag*** (or the near-duplicate ***UserVariant***) property to store a string or a unique value, and you wanted to search through the tree from a Node reference variable name *RootNode*. You could create your own drilling function and then access it like this:

```
Dim FoundNode As dynNode
'
' Find the node containing the tag
'
Set FoundNode = FindTag(RootNode, "Thing-A-Ma-Bopper")
'
' Report the results
'
If Not FoundNode Is Nothing Then    'FOUND IT!
  MsgBox "Found the tag in Node ID " & CStr(FoundNode.ID) & _
         ", Text Name: " & FoundNode.Text
Else                              'Drat! Not found!
  MsgBox "Did not find the Tag"
End If
```

The search routine, which is called ***FindTag*** in this example, could be defined as follows:

```
'*************************************************************************
' Function Name     : FindTag
' Purpose           : Sample function to drill down trough a tree, starting at the
'                   : specified node, and match the contents of a tag
'
' Inputs            : Node: A dynNode object to begin the search at
'                   : Tag:  The string data to search for
'
' Outputs           : Returns Nothing if the tag is not found, otherwise it returns a
'                   : reference to the Node that contains the matching tag
'                   :
'
' Assumes           : The Node is a valid dynNode object
'*************************************************************************
Function FindTag(Node As dynNode, Tag As String) As dynNode
'
' See if we have a local match
'
  If Node.Tag = Tag Then        'match?
    Set FindTag = Node          'yes, so return the found node
```

```
    Exit Function
  End If
'
' Else search through the node list of children, and recurse through them.
' If there are no child nodes, then the For-Next loop will not be processed
'
  With Node.Nodes
    Dim Index As Long
    For Index = 1 To .Count
      Set FindTag = FindTag(.Item(Index), Tag)       'do recursion (drill down)
      If Not FindTag Is Nothing Then Exit Function  'found a valid node match
    Next Index
  End With
End Function
```

As you can see, recursion actually makes for simple, cursive (pun) programming (the heading describing the function is almost as long as the code itself).

Example Of Drilling Down Through a Disk Directory and Duplicating it in Nodes

Following are a couple of simple methods that will drill down through a disk drive and collect its entire collection of files and folders using only a minimum of the class features we have discussed. ***Please note*** that to drill down through all of a large drive's folders and files can take several minutes (the same time it would take to do it *without* creating a node tree from it). The portions related to ***DynnamicNodes*** are marked in **Red**:

```
Private FSO As FileSystemObject     'standard system I/O object
                                    '(Windows Scripting Host Object Model reference)
Public RootNode As dynNode          'define new root node
'*************************************************************************
' Subroutine Name    : DrillDrive
' Purpose            : Enumerate the folder/files of a drive and set a reference
'                    : to the new node list in the public RootNode variable
'*************************************************************************
Private Function DrillDrive(DriveLetter As String) As dynNode
  Dim Drv As String

  Set FSO = New FileSystemObject            'Create system I/O object
  Drv = DriveLetter & ":\"                  'build a full file path
  Set RootNode = NewNodeList(Drv, Drv)      'create new root node (function defined in
moddynNode.bas)
  RootNode.Marker = True                    'mark it (we will mark all folders)
  'NOTE: this does not 'make' it a folder, but we are simply using the user-definable
  '      Marker property as a 'personal' reference.  This can be used for ANYTHING.
  EnumerateNodes RootNode                   'enumerate drive folders into it
  Set FSO = Nothing                         'release File I/O resources
End Sub

'*************************************************************************
' Subroutine Name    : EnumerateNodes
' Purpose            : Enumerate folders of drive into dynNode tree (recursive)
'
' Note that this routine calls itself repeatedly.  This is a technique called
' recursion, which is one of the coolest methods around for drilling through any
' branching-type system.  ...Unless you miss-code it and do not debug-step through
' it to ensure that it works, then the Vulcan Nerve Pinch (Ctrl-Alt-Delete with
' one hand) may be required.
'*************************************************************************
Private Sub EnumerateNodes(Parent As dynNode)
  Dim Fld As Folder     'Folder object from FSO classes
  Dim Fil As File       'File object from FSO classes
  Dim cNd As dynNode    'our local dynamic node
  Dim Ppath As String   'parent path (speed referencing by not using properties)

  Ppath = Parent.Key    'get parent path
'
' Build folders contained in Parent.  Use On Error Resume Next in case we
' encounter special system-level protected folder such as those on NT/XP
' platforms that will let you see them, but doe-nah thoucha it!
'
  On Error Resume Next                              'in case of special protected folders
  For Each Fld In FSO.GetFolder(Ppath).SubFolders 'get all folders
    Set cNd = Parent.Nodes.Add(, dynNodeChild, Fld.Path & "\", Fld.Name) 'add a node
    cNd.Marker = True                               'tag it as user-marked (folder)
    EnumerateNodes cNd                              'enumerate each folder
  Next Fld
```

```
'
' Build files contained in Parent (doing this into a TreeView node list would hang
' the system on my computer, as a TreeView is limited to 32,000 entries.  dynNodes
' allow up to 2,147,483,647 nodes (but who would be insane enough to try an make
' a tree list that long -- at least until personal computers are replaced by
' personal super-computers).
'
'SPECIAL NOTE!!!!:  Disable these following three lines (the For-Next loop) if you
'want to keep the wait down to a couple of minutes on a large system...
'
  For Each Fil In FSO.GetFolder(Ppath).Files
    Call Parent.Nodes.Add(, dynNodeChild, Fil.Path, Fil.Name)  'add a 'file' node
  Next Fil
End Sub
```