Understanding the TreeModel

Why 'Less Is More' Is an Elegant Design

By Eric Armstrong, Tom Santos, and Steve Wilson

In Swing, many components can be used with objects that implement corresponding models.

For example, Swing's JTree component can be used with an object that implements the TreeModel interface. Similarly, a JTable can be used with a TableModel, and a JList can be used with a ListModel.

But in each of these cases, a model interface is used mainly to define accessor methods. It defines no operations to manipulate the structure of the data -- no methods to insert or remove data items, or to change the order of data in the model.

That makes implementing a model and using it with one of these components somewhat different from using a more conventional MVC (model-view-controller) architecture.

But that difference allows greater flexibility. It means you can use JTree, JTable, or JList with an existing data structure without having to make violent modifications to to that structure to make it conform to the required interface.

This article focuses mainly on the relationship between JTree and the TreeModel interface. But similar thinking applies to using a JTable or a JList component.

The article comes with a downloadable sample program that shows exactly how JTree and TreeModel can be used together in a program.

TreeModel Basics

If you have an existing tree-based data structure, you can use it with a JTree without having to change your data model significantly. This is important to remember. There are many possible representations of tree structures, and there are many possible operations on them. But JTree makes no assumptions about which operations are available, so you are free to use an existing tree structure with a JTree -- including your operating system's directory tree! (For more information on this topic, see the article in this issue titled "*Creating TreeTables in Swing*.

The TreeModel Interface Defined

Although the JTree architecture defines a TreeModel interface, it defines no operations that affect the structure of the tree. Looking over the methods defined in the interface, you can see a variety of accessor methods, one method for changing the data associated with a node, and none at all for adding nodes, removing them, or shifting their position in the tree.

Here is how Swing defines the TreeModel interface:

```
public Object  getRoot();

public boolean isLeaf(Object node);
public int     getChildCount(Object parent);
public Object  getChild(Object parent, int index);
public int     getIndexOfChild(Object parent, Object child);
public void    valueForPathChanged(TreePath path, Object newValue);
void addTreeModelListener(TreeModelListener l);
void removeTreeModelListener(TreeModelListener l);
```

Turning a Weakness into a Strength

At first glance, that seems weird. How can a "model" leave out manipulation operations? But this apparent weakness quickly turns into a strength. The problem with defining a tree model is that there are many fully functional subsets of the possible operations on a tree. For example, let's say you define a tree model, and that you want to be able to add a sublist node either at the beginning or end of a node's sublist. There are at several possible combinations of methods that would achieve that goal. You could implement insert-first and insert-last, or you could implement insert-first and move-to-last, or you could implement append-to-end and move-to-first. Any of those pairs would be sufficient to make sure you can insert a new node either at the beginning or end of a list. The following table shows these three possible combinations of operations:

	Operation #1	**Operation #2**
1	Insert as first sublist node	Insert as last sublist node
2	Insert as first sublist node	Move node to end of its list
3	Append as last sublist node	Move node to beginning of its list

Any two of these operations are sufficient to allow the client-developer to get a node to the desired final positions. But which two should the tree model define? This kind of question arises repeatedly in the design of a tree object. Clearly, it would not be ideal for the tree model to define *every* possible operation on a tree. It would then be an onerous task for the developer to define a new model object, because of the many methods that would have to be defined.

But there is no "standard" subset of operations. The state of the art at the moment is that every component library which includes an MVC-based tree component defines a wildly different subset of the possible operations. The typical result, if you have an existing tree structure, is that you have to perpetrate extreme violence on it to make it a suitable model for a given component library.

Suppose, for example, that your structure implements "insert as first sublist" and "move node to end." If the component you plan to use defines its tree model using "insert as last sublist node" and "move node to beginning," you have your work cut out for you. You not only have to define the new methods required by the interface; you also have to redefine the old ones, to make sure they generate the events required by the view-component.

Like JList and JTable, JTree rushes to the rescue by the simple expedient of not defining *any* structure manipulations on the model. You are free to define any tree-structure operations that make sense, and therefore are free to use any existing tree object for the underlying data structure. The FileExplorer example that follows uses the tree structure defined by the operating system's directory hierarchy. The example shows you how to add listeners on the JTree component to find events of interest, manipulate the underlying data structure (in this case, the directory tree), and then fire off the events that tell JTree that the structure has changed.

Example: The FileExplorer Browser

To help you examine the principles and programming techniques described in this article, we've provided a sample program named *FileExplorer.java*, which implements a simple directory browser called FileExplorer. You can view the *FileExplorer.java* program as a series of text files, or you can download the program in an executable version, complete with all the source and class files that were used to compile it.

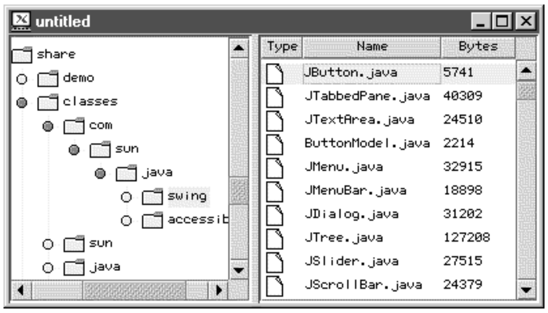
To see the source code for the various files that make up the FileExplorer sample program, follow these links:

- *FileExplorer.java*
- *AbstractTreeModel.java*
- *DirectoryModel.java*
- *DirectoryRenderer.java*
- *FileSystemModel.java*
- *FileSystemTreePanel.java*
- *TreeModelSupport.java*

You can also download a zipped file containing all the preceding source files, a *fileexplorer.jar* file, and an executable version of the program. To do that, follow this link:

FileExplorer.zip

When you excecute the *FileExplorer.java*program, it displays a FileExplorer browser has a left pane and a right pane, as shown in the following illustration. The left pane contains a tree-structured file hierarchy. It shows all files in a directory, as well as any folders that the directory contains. When a folder is selected in the left pane, the right pane shows a table of information about the files in the directory.



NOTE: *The FileExplorer is a demonstration application that has been kept as simple as possible. To make a real utility out of it, you would have to add many additional trimmings.*

Implementing the TreeModel interface

The first step in using JTree with an existing model is to implement the `swing.tree.TreeModel` interface. You can add that interface to an existing class, implement it on an adapter that delegates to the existing tree structure (or subclasses that data object), or use the interface on a "parallel" tree model object that simply reports changes you make to the real tree. But, however you slice it, you need to implement that interface.

NOTE: *The FileExplorer uses the "parallel" approach because that strategy lets you create an adapter that bridges the tree model and the underlying data tree (in this case, the directory structure) without having to*

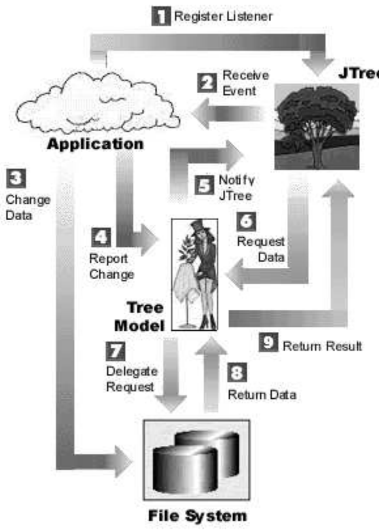
modify any of the code that works with the underlying data tree.

Creating a JTree object

After creating a model, you instantiate a JTree object that uses the model by invoking the `JTree(TreeModel)` constructor. Then you register event listeners with the JTree component to handle the events you are interested in. When the program handles a user event, it makes two calls: one to the underlying data structure to make the change, and another to the TreeModel-adapter to report the change. The TreeModel object then notifies the JTree of the change.

The diagram on the right shows the steps needed to create a JTree object. They work like this:

1. The app registers mouse and keystroke listeners with the JTree.
2. The application receives events from the JTree and determines the action to take.
3. The application changes the underlying data model (in this case, the file system).
4. The application reports the change to the TreeModel adapter.
5. The TreeModel adapter notifies its listeners (JTree) of the change.
6. JTree asks the TreeModel adapter for data to display.
7. The adapter delegates the data request.
8. The underlying data object (file system) passes back the data.
9. The adapter sends the data back to the JTree.



The remainder of this article describes these steps in detail, showing how to use a JTree with an existing tree data structure. In this case, the FileExplorer example included with the Swing class library uses a JTree to display the directory tree managed by your operating system.

NOTE: *You get a cleaner design if the TreeModel adapter does the work of changing the data. But this diagram*

serves to illustrate the very important point that changing the data and notifying the JTree of the component of the change are two very distinct steps.

TreeModel requirements

There are two aspects of the TreeModel interface that you need to pay attention to: the *explicit* requirements and the *implicit* requirements. The explicit requirements are the methods defined in the interface. You saw those in the previous section. The JTree component uses those methods to access the model's data, to change the data stored at a node, and to register a TreeModel listener -- which defines the implicit requirements for the interface.

When the JTree component is created, it registers itself with the model as a `TreeModelListener`. Implicitly, then, the TreeModel is expected to perform the following notifications, as defined in `swing.event.TreeModelListener`:

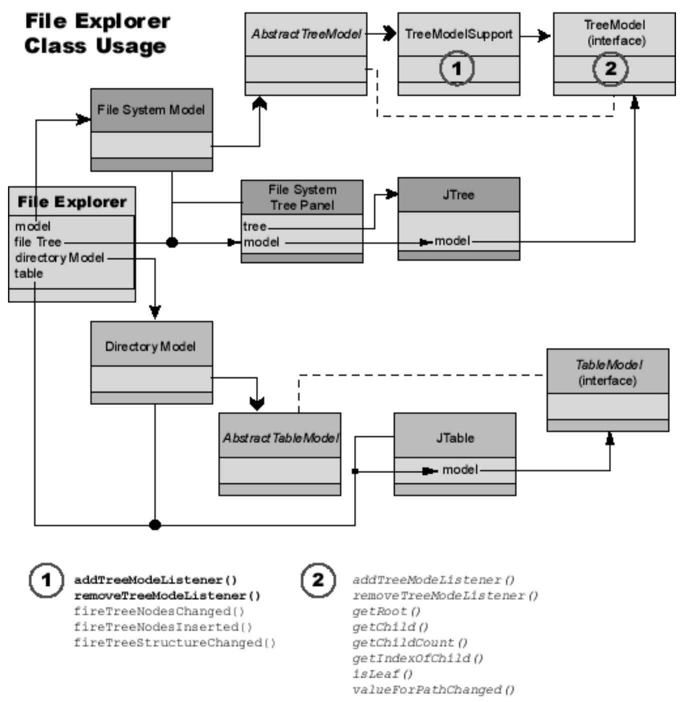
```
void treeNodesChanged(TreeModelEvent e);

void treeNodesInserted(TreeModelEvent e);

void treeNodesRemoved(TreeModelEvent e);

void treeStructureChanged(TreeModelEvent e);
```

As you can see, the TreeModel is expected to generate events when a nodes are changed, inserted, or removed, as well as when more global structure changes occur. To encapsulate the event information, the model needs to generate `swing.event.TreeModelEvent` objects.

Using AbstractTreeModel

To minimize the work you need to do, you can subclass `swing.tree.AbstractTreeModel`. This class extends `swing.tree.TreeModelSupport`, which handles the registration of tree model listeners and event notifications. Since it also declares the `TreeModel` interface, leaving you to implement the methods in that interface (except for the listener registration methods, which are already handled by `TreeModelSupport`.)

The following class diagram shows these relationships. Since FileExplorer also uses a JTable to display the files in a directory, the diagram shows the parallel relationships that exist in the use of the JTable component.



As this diagram illustrates, `AbstractTreeModel` inherits concrete implementations of `addTreeModelListener` and `removeTreeModelListener` from the `TreeModelSupport` class, which takes care of the listener-registration methods required by the `TreeModel` interface. The `TreeModelSupport` class also provides the event-notification methods `fireTreeNodesChanged`, `fireTreeNodesInserted`, `fireTreeNodesRemoved`, and `fireTreeStructureChanged`.

That leaves the accessor methods to be implemented by the application-specific model, `FileSystemModel`: `getRoot`, `getChild`, `getChildCount`, `getIndexOfChild`, and `isLeaf`. If the tree is editable, the application-specific model must also provide semantics for `valueForPathChanged`. Otherwise, this method can be given a null-implementation.

NOTE: Because the `TreeModelSupport` class is separate from `AbstractTreeModel`, you can either subclass `AbstractTreeModel` to create an adapter, or add the `TreeModel` interface on an existing class and then delegate the registration and notification behaviors to a `TreeModelSupport` object.

The `FileExplorer` class creates an instance of `FileSystemModel`, which it passes to an instance of `FileSystemTreePanel`. The object-passing is shown in the diagram by the black dot, which indicates that an object of the class `FileSystemModel` is passed to the `FileSystemTreePanel`. The arrow at the target end shows where the `FileSystemTreePanel` keeps the passed object (in the `model` attribute). The `FileSystemTreePanel`, in turn, creates a `JTree` object, and passes its `model` object to the `JTree` instance. As shown in the diagram, the type of the `JTree`'s `model` object is `TreeModel`, which completes the circle -- the `FileSystemModel` object is created from a class that implements `TreeModel`, and is ultimately stored in a variable defined with that type.

The tree view is on the left side of the `FileExplorer` app. On the right side is the table of files and their properties, which follows the same pattern: The `FileExplorer` class creates an instance of `DirectoryModel`, which subclasses `AbstractTableModel`. This `TableModel` object is then passed to an instance of `JTable`.

Identifying Leaf Nodes

In a tree like this one, where some nodes (directory nodes) can have children and other nodes (file nodes) cannot, `isLeaf()` is implemented to distinguish the kinds of nodes. In this case, the nodes are cast to type `File`, and `isFile()` is returned to identify leaf nodes. Here is the code for `isLeaf()`:

```
public boolean isLeaf( Object node ) {

    return ((File)node).isFile();

}
```

In a tree where any node can have children, however, `isLeaf()` could simply return `true`.

Handling JTree Events

When a directory in the left pane is clicked, the app needs to change the table of information displayed in the right pane. To do that, registers a tree selection listener, like this:

```
fileTree.getTree().addTreeSelectionListener

              ( new TreeListener( directoryModel ) );
```

The DirectoryModel object is passed to the TreeListener for use in the update process. Here is the code for the DirectoryModel class:

```
protected static class TreeListener implements TreeSelectionListener {

    DirectoryModel model;
    public TreeListener( DirectoryModel mdl ) {
        model = mdl;
    }
    public void valueChanged( TreeSelectionEvent e ) {

        if (e.getNewLeadSelectionPath() == null) return;

        File fileSysEntity = (File)e.getPath().getLastPathComponent();
        if ( fileSysEntity.isDirectory() ) {
            model.setDirectory( fileSysEntity );
        }
        else {
            model.setDirectory( null );
        }
    }
}
```

This code identifies the `File` object that was selected by the user. (The `getPath()` method returns a `TreePath` object, which contains an array of `Object`s. The first object in the array comes from the root of the tree. The last object in the array is the selected item. The items between the root and the selected item identify the path to the object, in the same way that directory names identify a path to a file. (In this case, since the tree represents the system file structure, the path objects *are* directories.)

NOTE: The `valueChanged()` method reports changes in the current selection. As a result, `getNewLeadSelectionPath()` can return `null` whenever the currently selected item has been deleted. A second event is then generated which identifies the new selection after the delete takes place. Although the FileExplorer app does not handle deletes, you should get in the habit of coding

```
if (e.getNewLeadSelectionPath() == null) return;
```

at the start of a `valueChanged()` method for TreeSelectionEvents.

The `getLastPathComponent()` method returns the last object in the path, which is then cast to a `File` object. If the `File` object is a directory, it is sent to the `DirectoryModel` for display. Otherwise, the `DirectoryModel` is messaged with `null`, which clears the display in the right pane.

NOTE: When the user has changed the current selection, but has not deleted a node, the TreeSelectionEvent contains the difference in the selection. In other words, it contains old nodes that are no longer selected as well as newly selected nodes.When you are concerned with multiple-node selections, you can invoke the TreeSelectionEvent method `isAddedPath(TreePath)` on each of the items returned by its `getPaths()` method to find out which were added and which were removed. Or you can use the JTree `getSelectionModel()` method to obtain the object which is maintaining the current selection list, and then use its `getSelectionPaths()` method to get an array of selected TreePaths. Simple apps which aren't concerned with multiple-node selections (like FileExplorer) simply use the first selected node, which is returned by the TreeSelection event `getPath()` method.

Reporting Changes to the Model

The `FileExplorer` example does not currently make any changes to the file system. Some useful extensions to it might allow typing to on a filename to rename the file, or dragging a file to a new location. After the changes are made to the underlying file system, the JTree view needs to be notified. To do that you create a `TreeModelEvent` object and then notify listeners, passing the event object as data.

Once again, the easiest way to notify listeners is to use one of the methods defined in the `TreeModelSupport` class, `fireTreeNodesChanged`, `fireTreeNodesInserted`, `fireTreeNodesRemoved`, or `fireTreeStructureChanged`. These methods invoke the appropriate method in each of the registered listeners in order to notify them of the change, where the appropriate method is one of the following:

```
void treeNodesChanged(TreeModelEvent e);

void treeNodesInserted(TreeModelEvent e);

void treeNodesRemoved(TreeModelEvent e);

void treeStructureChanged(TreeModelEvent e);
```

The only thing left to understand is how to create a TreeModelEvent. Depending on the type of notification you are making, you will use one of the two types of TreeModelEvent constructors shown here:

```
public TreeModelEvent(Object source, Object[] path,

       int[] childIndices, Object[] children)

public TreeModelEvent(Object source, TreePath path,

       int[] childIndices, Object[] children)

---------------------------------------------------

public TreeModelEvent(Object source, Object[] path)

public TreeModelEvent(Object source, TreePath path)
```

Note that there are essentially two kinds of constructors -- one type specifies child nodes, the other doesn't. Within each type, you have a choice of specifying either a TreePath or an array of Objects to specify a target node. But the important distinction is whether or not you specify children. The following list summarizes the rules you need to know to create the right tree model event:

1. When making a nodes-changed, nodes-inserted, or nodes-removed notification you *always* specify children. So you use one of the top two constructors.

2. Only when you are making a structure-changed notification do you use the simple constructor that does not specify children.

3. In all cases, the `path` argument points to the *parent* of the changes. If nodes were inserted, the path points to the parent node under which the inserts took place. Similarly for deletes or changes.

4. Since children are specified as indexes under a single parent, it follows that a single insert/delete/change notification only covers one node's sublist. Changes to multiple lists require multiple notifications.

5. In order to notify a listener of multiple inserts and deletes, or to identify changes at multiple levels in the tree, you use `treeStructureChanged()` and use the simple, no-children TreeModelEvent. In this case, the path argument specifies

a node in the tree that did *not* change, and which has all of the other changes below it.

6. When specifying changes, the indexes specify the list positions which will be replaced with the specified children Objects, so they become the new children of the parent node specified by *path*. In this case, the order of the indexes is immaterial, but the usual practice is to specify them from low to high.

7. When specifying inserts, the indexes specify the list positions in the *final* list, after the inserts have taken place. The indexes *must* be specified from lowest to highest.

8. When specifying deletes, the indexes specify the list positions in the *initial* list, before the deletes have taken place. Again, the indexes *must* be specified from lowest to highest. (Tree model listeners like JTree process them from back to front, so that one delete does not change the index position of another deleted item.)

For more information on this subject, see the TreeModel and TreeModelEvent APIs. The API comments in these modules describe the process in even greater detail.

Using the Default Tree Model

Now that you have seen how to add a JTree view to an existing tree structure, you may be interested in knowing how to use the default tree model to create a new tree structure. The remainder of this write-up discusses that process.

Unfortunately, although JTree creates a default tree model when you use the null constructor `JTree()`, its not so easy to add nodes to that model or manipulate it in other ways. To do so, you first obtain the default model currently in use by coding:

```
DefaultTreeModel t = (DefaultTreeModel) myTree.getTreeModel();
```

The cast from the `TreeModel` returned by `getTreeModel()` to a `DefaultTreeModel` succeeds as long as the JTree was created with the null constructor or with any of the constructors that specify an `Object` value, an array of `Object`s, a `TreeNode`, or a `Hashtable`. Each of those constructors creates an instance of `DefaultTreeModel` wrapped around the specified object(s). The cast may fail, however, if the `JTree(TreeModel)` constructor was used to create a `JTree` with something other than the default tree model..

Once you have a DefaultTreeModel object, you can use the following TreeModel methods to access specific data values:

```
public Object  getRoot();

public int     getChildCount(Object parent);

public Object  getChild(Object parent, int index);

public int     getIndexOfChild(Object parent, Object child);
```

All of these methods *return* data values, however. You want to add new nodes and do other structure maipulations. There are basically two kinds of things you might want to do. You might want to add nodes to an otherwise static tree, building it one time in order to display it, or you might want to do ongoing modifications to the tree structure dynamically. Those two cases are discussed below.

Adding Nodes to a Static Tree

Its fairly easy to create a static tree for JTree to display. To do that, you create a tree structure using `DefaultMutableTreeNode` objects and then pass the root of that tree to a JTree constructor. Here is how the SwingSet demo program builds up a tree structure by creating multiple `DefaultMutableTreeNode` object:s and linking them together:

```
DefaultMutableTreeNode root = new DefaultMutableTreeNode("Music");
DefaultMutableTreeNode category;
DefaultMutableTreeNode composer;
DefaultMutableTreeNode style;
DefaultMutableTreeNode album;

// Classical
category = new DefaultMutableTreeNode("Classical");
top.add(category);

// Beethoven
category.add(composer = new DefaultMutableTreeNode("Beethoven"));
composer.add(style = new DefaultMutableTreeNode("Concertos"));
style.add(new DefaultMutableTreeNode("No. 1 - C Major"));
```

```
...
composer.add(style = new DefaultMutableTreeNode("Quartets"));
style.add(new DefaultMutableTreeNode("Six String Quartets"));
...

JTree tree = new JTree(root);
```

Note that object was needed for each level of the tree (category, composer, and so on) so that nodes could be added to the list created at that level. For example, the line

```
category.add(composer = new DefaultMutableTreeNode("Beethoven"));
```

adds a new composer object under the current category and the same time maintains a reference to the composer object so that nodes can be added under *it*. This is a good model to follow to create and display a simple tree structure.

Dynamically Modifying a Tree

The `add()` method in the `DefaultMutableTreeNode` class is good for appending items to the end of a list. This class defines many other useful operations, including:

```
getParent() insert(node, index)

getNextSibling() removeFromParent()

getPreviousSibling() removeAllChildren()

getFirstChild() removeChild(index)

getChildAfter(node) removeChild(node)
```

The get-methods in this list all return `DefaultMutableTreeNode` objects. So once you have one, you're golden. The trick, though, is to get the first `DefaultMutableTreeNode`. The answer lies in knowing that a `DefaultTreeModel` created by the zero-argument `JTree()` constructor is, in fact, composed of `DefaultMutableTreeNode` objects.

You can get the root node from a `DefaultTreeModel`, `t`, using:

```
DefaultMutableTreeNode root =
    (DefaultMutableTreeNode) t.getRoot();
```

Once again you must cast the result, since `getRoot()` returns an `Object` (which happens to be stored in the tree model as a TreeNode). But since that object is really a `DefaultMutableTreeNode`, the cast succeeds.

NOTE: The cast works as long as you are using a `JTree` that was created with the `DefaultTreeModel`. If the `JTree` was creating using the `JTree(TreeModel)` constructor and some class that does not store `DefaultMutableTreeNode` objects, the cast will fail.

However, when you make a change using a `DefaultMutableTreeNode`, you also need to inform the `DefaultTreeModel` that it has changed! To that, you use the following `DefaultTreeModel` methods:

```
reload()

nodesWereInserted()

nodesWereRemoved()

nodesChanged()

nodeStructureChanged()
```

Since `DefaultMutableTreeNode` objects implement `swing.tree.MutableTreeNode`, you can avoid the two-step process of making changes *and* notifying the model of them by using the following methods, which make changes directly to the `DefaultTreeModel`:

```
public void insertNodeInto(MutableTreeNode newChild,

                           MutableTreeNode parent, int index){

public void removeNodeFromParent(MutableTreeNode node) {
```

To do anything interactive, you will need to intercept user events and make changes directed by the user. To get the `DefaultMutableTreeNode` associated with a given mouse click you will need to use the JTree method `getPathforLocation()` or `getClosestPathForLocation()`, either of which returns a `TreePath` object. The `TreePath` objects defined by the `DefaultTreeModel` consist of an array of `DefaultMutableTreeNode` objects, where the first object is the root, and the last is the target node. To get the `TreeNode` object for the target node, you need code like this:

```
TreePath p = JTree.getPathForLocation(x,y);

DefaultMutableTreeNode node =

          (DefaultMutableTreeNode) p.getLastPathComponent();
```

This code gets the tree path object for a pair of x,y coordinates. It then gets the last element in that path and casts it to a `DefaultMutableTreeNode`, which you can then use for structure manipulations. Again, remember to inform the model if you make changes directly from the node!

Summary

If you have an existing tree structure, it is easier to attach a JTree view to it than it use to construct a JTree and use its default tree model. When you are attaching to an existing tree structure, you don't have to worry about `TreeNode`, `MutableTreeNode`, or `DefaultMutableTreeNode` objects. Instead, you can ignore all that and concentrate on the data objects in your structure. So it's easy to create a static tree, and it's easy to add a JTree view to an existing tree. When you want to create a default tree and manipulate it, however, the process can be tricky. So be careful.