The Tree Control: Managing Data with JTree

By Kim Topley

Date: Apr 12, 2002

Sample Chapter is provided courtesy of Prentice Hall.

Return to the article

This chapter takes an in-depth look at JTree, one of the two complex Swing components that manage collections of related data. Kim Topley shows you how to use many of the tree's powerful features and how to extend or use them directly in your own applications. Also learn how to use JTree to render individual display elements to allow the user to directly edit the tree's data.

Topics in This Chapter

- Creating and Working with Trees
- A File System Tree Control
- Creating Customized Tree Renderers
- Trees and Tooltips
- Tree Editing and Customized Editors

This chapter takes an in-depth look at `JTree`, one of the two complex Swing components that manage collections of related data. `JTree` is intended to represent information that has a hierarchical structure, of which the prototypical and probably most familiar example might be a file system. You'll see how simple it is to create a basic `JTree` and look at the ways in which the user can interact with the control to expose and hide data and to have the application operate on the objects that the tree's content represents.

A large part of this chapter is devoted to developing a control that allows you to display the contents of a file system in the form of a tree. While building this control, you'll learn how to use many of the tree's powerful features and, when it's finished, you'll have a component that you can extend or use directly in your own applications.

Finally, `JTree` allows you to take over the job of rendering individual display elements. The second half of this chapter looks at how this can be achieved and how to allow the user to directly edit the tree's data.

The Tree Control

The `JTree` component is the first of two controls that you'll meet in this book that are intended to present views of large amounts of data. The difference between `JTree` and the other control, `JTable`, is that the former is well-suited to handling data that is hierarchical in nature, while the latter is used with information that can be organized into rows and columns. The most familiar example of a tree control is probably the one in the left pane of Windows Explorer, which shows a directory view of a file system. Later in this section, you'll see how to implement a Java version of this control, which is pictured in Figure 10-1.

Figure 10-1 A tree showing a file system.

Tree Nodes

All trees start with a *root node*, which is customarily shown at the top left of the control. By default, when you create a `JTree`, the root is visible, together with the objects in the level immediately below it. Figure 10-1 shows a view of part of the `C:` drive of my laptop. In this case, the `C:\WINDOWS` directory is the root and the directories that reside within it are shown below and to the right of it. The Metal, Motif, and Windows look-and-feel classes all represent each node in the tree using both an icon and a text string. In the case of a file system, the natural way to use the text string is to display the name of the directory that corresponds to the node, while the icon shows either an open or closed folder, depending on whether the contents of the directory are visible or not.

In Java 2 version 1.3, `JTree` is sensitive to its component orientation so that, in a right-to-left locale, the root node will appear at the top right instead of the top left and the tree expands from right to left. Within each node, the left-to-right order of the expansion handle, the icon and the text is reversed, producing a complete mirror image.

Core Note

The descriptions that are given in this section, unless otherwise qualified, relate to the appearance of the control when the Metal look-and-feel is installed. Installing a different look-and-feel can have a radical effect on the way the tree looks and there can even be variations within a single look-and-feel, as you'll see.

It is simple and routine to change the text string that describes the node. Changing the icon is a topic that will be left until the end of the chapter, so most of the examples will have the usual folder representation, whether or not they represent a file system.

Tree nodes may or may not have other nodes, known as child nodes, linked beneath them. Each node that has child nodes can be considered to be the root of a subtree. For example, in Figure 10-1, the node `C:\WINDOWS\SYS-TEM` has several child nodes beneath it and can be thought of as the root of the subtree composed of itself and those nodes.

Nodes that are at the same level of the tree are displayed in a vertical line so that, in the case of a file system tree, all of the directories directly below the root are shown one above the other. As you move further to the right on the screen, you are moving further down the file system hierarchy, until you reach a node that doesn't have any nodes attached below it. Such nodes are often called leaf nodes, because they reside at the very end of a sequence of branches (otherwise known as branch nodes) that join nodes to each other. In the case of `JTree`, however, a node that doesn't have any nodes below it need not be a leaf node, as you'll see. As far as the tree is concerned, whether or not a given node is a leaf matters only so far as it affects the icon used to represent it—depending on the look-and-feel, leaf nodes are typically represented by a folded sheet of paper instead of the folder used to represent the other nodes.

The tree may be configured to join its child nodes with lines to make the hierarchy easier to see. Whether this is possible depends on the look-and-feel in use. When this option is enabled, nodes at the same level are typically connected to each other by a vertical ine as shown in Figure 10-1; each node at that level connects to the vertical line with a small horizontal dashed line. If the node does not have any nodes below it, the horizontal and vertical lines just meet at a point, as is the case with the node `C:\WINDOWS\SYS-TEM\DTCLOG` in Figure 10-1. At the intersection point for a node that does have other nodes below it in the hierarchy is a small control known as an expansion handle, which may be rendered differently depending on whether these nodes are visible. Assuming these nodes are not visible, clicking on the expansion handle makes the tree expand to the right to show the next level of nodes, and downward to make room for the nodes that have just become visible to be stacked vertically. In Figure 10-1, the node `C:\WINDOWS\SYSTEM` has been expanded in this way to show its child nodes. Notice that the node `C:\WINDOWS\WEB` has moved downward to make room for the children of `C:\WINDOWS\SYSTEM`, and that the expansion handle to the left of the expanded node has been drawn with the "handle" part pointing downwards, whereas the handle for nodes that have not been expanded points to the right. Clicking on the expansion handle when the node is already expanded will cause the child nodes to disappear and the nodes at `C:\WINDOWS\WEB` and below to move upward to occupy the free space. There is never more than one node on any given row.

Tree Appearance

To emphasize how dependent the appearance of a tree and its nodes are on the selected look-and-feel, let's look at some of the possible variations. Figure 10-1 shows a tree with the Metal look-and-feel selected. The same tree in the Windows look-and-feel is shown in Figure 10-2, while the Motif version can

be seen in Figure 10-3.

It's pretty evident that there are differences between these figures and Figure 10-1. As you can see, Windows and Motif use boxes to indicate nodes that have children. When the children are invisible, the expansion box has a plus sign to indicate that there is more of the tree to be viewed; when you open such a node to show its children, the plus sign changes to a minus sign, as you can see in the case of the `C:\WINDOWS\SYSTEM` node. The icons are also subtlly different and the line styles vary between look-and-feel implementations.

It is even possible to have different representations of the same tree within a single look-and-feel. The tree has a client property called `lineStyle` that is used to control the lines drawn between nodes. This property is currently supported only by the Metal look-and-feel, but there is nothing about the mechanism used to implement it that binds it to a single look-and-feel, so you may find it implemented more widely in the future. The `lineStyle` property has three possible values, listed in Table 10-1.

Table 10-1 LineStyle Settings

Setting	Effect
`None`	No lines are drawn anywhere on the tree.
`Angled`	Vertical lines between nodes at the same level, horizontal line to child node.
`Horizontal`	A single horizontal line between nodes immediately attached to the root node. This is the default setting.

Figure 10-1 shows a tree with the `Angled` setting, which is the most natural way to represent a file system and is most like the appearance of Windows and Motif trees. This configuration can be obtained using code like this:

```
Jtree tree = new Jtree() ;
```

Figure 10-2 A tree as drawn by the Windows look-and-feel

Figure 10-3 A Motif tree.

```
Tree.putClientProperty("Jtree.lineStyle", "Angled");
```

Similar code is required to obtain the other two possible effects. The same tree rendered with `lineStyle` set to `None` is shown in Figure 10-4 and with `lineStyle` set to `Horizontal` in Figure 10-5. Because `None` is the default, all of your trees will look like the one in Figure 10-4 if you don't explicitly select a `lineStyle`. Since `lineStyle` is ignored by look-and-feel implementations that don't provide it, it is always safe to set a specific value no matter which look-and-feel is selected.

Tree Node Selection

The mouse can be used to select an item from the tree by clicking on the icon or the text. Clicking twice on

either of these is the same as clicking on the expansion handle—the node either expands or collapses depending on its current state. The ability to make a selection is one of the main reasons for using a tree control. As you'll see, you can attach listeners that receive notifi-cation when a selection is made and you can also arrange to be notified when part of the tree is expanded or collapsed or is about to do so.

Figure 10-4 A Metal tree with line style "None."

Elements of the Tree

Now let's turn to how a tree is represented in terms of data structures. The only elements of any substance in a tree are the nodes themselves—the way in which these nodes relate to each other is represented by references from one node to another. The `JTree` control doesn't directly manage the nodes themselves, or remember how they are organized. Instead, it delegates this to its data model.

As with the other Swing components, `JTree` deals mainly with generic objects that are represented in Java by an interface. For example, any class can be used as a data model for the tree, so long as it implements the `Tree-Model` interface and any class can participate in the model as a node in the tree so long as it implements the interface `TreeNode`. This design pattern has been shown before in connection with `JComboBox` and `JList`, which have data models specified as interfaces and concrete implementations of those interfaces that are used by default. The `JTree` (and, to some extent `JTable`) is a little different from `JComboBox` and `JList` in that it is not really good enough to just implement the `TreeNode` and `TreeModel` interfaces. The default implementations of these interfaces in the Swing `tree` package provide facilities far beyond those specified by `TreeNode` and `TreeModel`. Furthermore, in many cases, these extra facilities can be accessed directly through the model or indirectly via the `JTree` object and there will be a loss of functionality if classes that only implement the minimal interface are used instead of the default ones. In practice, tree nodes are most likely to be instances of the class `DefaultMutableTreeNode`, while the model will probably be derived from `DefaultTreeModel`. For specific, lightweight uses of the tree where the full functionality of `DefaultMu-tableTreeNode` and `DefaultTreeModel` are not required, it might be useful to develop less functional implementations.

Figure 10-5 A Metal tree with line style "Horizontal."

`JTree` is unlike `JComboBox` and `JList` in another way. When you create either of the latter two, you populate the model and then wait for a selection. The item that is retrieved from the control as the user's selection is of the same type as the data held within the model itself. For example, a list box populated with `String`s would return a selected item of type `String`. With `JTree`, things are not so simple. The model deals in terms of nodes, but the user interface to the tree uses a class called `TreePath` and the immediate result of a selection is one or more `TreePath`s. As you'll see later, there is a way to map from a `TreePath` to the corresponding `TreeNode`, which means that it is possible, from an event listener, to get back to the original node in the data model. This node is not, however, of any direct use to the selection event handler unless some information has been stored with the node that is of meaning to the user of the tree. Suppose, for example, that you construct a tree that shows the content of a file system in a file open dialog and the user double-clicks on an item to have it returned to the tree user as the selected file. There are two ways that the interface to the user of this tree could be built—either the name of the selected file would be returned, or alternatively you could choose to return an object that represents the file that was stored with the file's node when the tree was created. The second of these alternatives is obviously more powerful. To make such an implementation possible, the tree nodes allow you to store a reference to an arbitrary object, called the *user object,* with each node. How you use this is up to you—you might just choose to make it the file's name, or it might be a `java.io.File` object for the file.

Creating a Tree

`JTree` has no less than seven constructors, most of which allow you to initialize the data model with a small quantity of data taken from the more common data collection classes in the JDK:

```
public JTree();
public JTree(Hashtable value);
public JTree(Vector value);
public JTree(Object[] value);
public JTree(TreeModel model);
public JTree(TreeNode rootNode);
public JTree(TreeNode rootNode,
             boolean askAllowsChildre);
```

The first constructor gives you a tree with a small data model already installed; this can be useful when you write your first program that uses a tree, but after that you'll probably never use it again. The next three constructors initialize the tree from a `Hashtable`, a `Vector`, and an array of arbitrary objects. Since all of these are flat data structures, the resulting tree is also flat, consisting of a root node and the data from the object passed to the constructor attached to it, with one node per entry in the table, vector or array. Figure 10-6 shows a tree created from a `Hashtable` containing five entries.

Figure 10-6 A tree created from a Hashtable.

The code that was used to create this tree is shown in Listing 10-1 and can be run using the command

```
java JFCBook.Chapter10.HashHandleTree.
```

Listing 10-1 Creating a Tree with a Hashtable

```
package JFCBook.Chapter10;

import javax.swing.*;
import javax.swing.tree.*;
import java.util.*;
public class HashHandleTree {
  public static void main(String[] args) {
    JFrame f = new JFrame("Tree Created From a Hashtable");
    Hashtable h = new Hashtable();
    h.put("One", "Number one");
    h.put("Two", "Number two");
    h.put("Three", "Number three");
    h.put("Four", "Number four");
    h.put("Five", "Number five");
    JTree t = new JTree(h);
    t.putClientProperty("JTree.lineStyle", "Angled");

    t.setShowsRootHandles(false);
    f.getContentPane().add(t);
    f.pack();
    f.setVisible(true);
  }
}
```

As you can see from Figure 10-6, the tree displays all of the items from the `Hashtable`. Each item has been turned into a leaf node and added directly beneath the root node of the tree. However, since a root node was not explicitly supplied, the tree doesn't display one. This is a common feature amongst trees created with the second, third, and fourth constructors. A consequence of this is that you can't collapse this tree. You can force the root node to be displayed by using the `JTree setRootVisible` method:

```
t.setRootVisible(true);
```

This would show the root node as an open folder labeled `root`, with an expansion icon to the left of it. If you don't want to show the expansion icon, you can disable it using `setShowsRootHandles`:

```
t.setShowsRootHandles(false);
```

Figure 10-7 shows two trees, both with the root node visible. The left-hand tree was created with `setShowsRootHandles(true)` and the right one with `setShowsRootHandles(false)`.

Figure 10-7 A tree with the root visible.

You can expand or collapse the tree by double-clicking on the root folder or its label. You can also use the keyboard to navigate the tree and open or close nodes. If you select the root node, you can use the down arrow key to move down the set of child nodes and the up key to move back up again. When a node that has children is selected and closed, the right arrow key will open it. Similarly, you can use the left arrow key on an open branch node to close it.

In cases like this where the root node was created automatically, the arbitrary label `root` is assigned to it. You can, if you wish, change this label or remove it entirely. You'll see how to do this later.

You'll notice that the items in the tree are not ordered in a particularly sensible way. This happens because `Hashtable`s don't maintain the order of the items that you place into them. The tree is constructed by getting

an enumeration of the items in the `Hashtable`, which doesn't guarantee any particular ordering. If you care about the order in which the data is displayed, you should use a `Vector` or an array.

Most trees will be created using one of the last three constructors, which require a complete tree model or at least the root node to have already been constructed. If you choose to supply a model, you can use the Swing `DefaultTreeModel` class, or create your own as long as it implements the `TreeModel` interface. For most purposes, `DefaultTreeModel` is more than adequate and, as mentioned earlier, it offers useful facilities over and above the basic interface, many of which will be used in this chapter.

The TreeNode Interface

Whether or not you explicitly supply the model, you will need to create the root node and attach to it all the data for the tree. All nodes are based on the `TreeNode` interface, which has seven methods:

```
public Enumeration children()
public boolean getAllowsChildren()
public TreeNode getChildAt(int index)
public int getChildCount()
public int getIndex(TreeNode child)
public TreeNode getParent()
public boolean isLeaf()
```

With the exception of `isLeaf`, all of these methods are concerned with the relationship between a node and its parent and children. Every node (apart from the root node) has exactly one parent and may have any number of children. If the node is a leaf node, it doesn't have children. It is also possible to have a type of node that shouldn't be considered to be a leaf when the tree draws its on-screen representation, but still doesn't allow child nodes to be connected to it. Nodes with this property return `false` to the `getAllowsChildren`, as should any node that can only be a leaf (such as a node representing a file in a file system). Child nodes are ordered within their parent, because the ordering might be important to the data that is being represented. Because of this property, it is possible to get the `TreeNode` object for a child at a given index within the parent's collection (using `getChildAt`) or the index for a particular child node (from `getIndex`). The `getParent` method returns a node's parent node. In the case of the root node, this returns `null`. Finally, `getChildCount` returns the number of children directly connected to a node. This value does not include grandchildren and more distant descendants.

As you can see, `TreeNode` is a read-only interface. To make changes to a node or its position in the tree, you need an instance of `MutableTreeNode`, an interface that extends `TreeNode`. Usually, the nodes that you actually create will be instances of `DefaultMutableTreeNode`, which is the Swing class implementing `MutableTreeNode`, so that it is usually possible to get a writable reference to any node even if you're only given a `TreeNode` reference by casting this reference to a `MutableTreeNode`. If you don't have prior knowledge of the type of node you are dealing with (perhaps because you're writing code to drive a third-party tree-based component), you should check that your `TreeNode` is an instance of `MutableTreeNode` before doing this.

The MutableTreeNode Interface

The `MutableTreeNode` interface adds the following methods to `TreeNode`:

```
public void insert(MutableTreeNode child, int index)
public void remove(int index)
public void remove(MutableTreeNode node)
public void removeFromParent()
public void setParent(MutableTreeNode parent)
public void setUserObject(Object userObject)
```

By contrast to `TreeNode`, these methods are entirely concerned with creating and changing the parent-child relationships between nodes. The `insert` method adds the given node as a child of the node against which it is invoked, specifying the index that should be used for the child. `DefaultMu-tableTreeNode` provides a more convenient way of adding children that doesn't require you to keep track of indices, as you'll see shortly. The `remove` methods disconnect a node from its parent given either the node reference or an index. Whichever variant of `remove` you use, you must invoke this method against the *parent* of the node that you want to remove. To remove a child node without a reference to its parent, invoke `removeFromParent` against the node itself.

The `setParent` method sets the reference from a child to its parent. You are very unlikely to want to use this

method, however, because it doesn't make the parent itself aware of the child. This method is really intended for use in the implementation of the other `MutableTreeNode` methods, such as `insert`.

The last method in this interface is `setUserObject`, which associates an arbitrary object with the node. This is, in fact, the only way to make a node useful. The other methods of `MutableTreeNode` deal with the construction of the tree, but this method allows you to say what the objects in the tree represent. In the examples shown above, the strings `One`, `Two`, `Three` and so on were the useful content of the tree (although "useful" might not be the right word in the case of this trivial example). They are stored in the tree as the user objects of the nodes that represent them. If you use the `DefaultMu-tableTreeNode` class to create your nodes, you supply the user object to the constructor, and you can also use the `setUserObject` method to change the associated object later.

Now let's look at the `DefaultMutableTreeNode` class which, as has been said, implements a superset of the `MutableTreeNode` interface. Unless you're implementing a very lightweight tree and have special requirements, it's not worth considering implementing your own `MutableTreeNode` class. All of the examples in this chapter will directly use or derive from `Default-MutableTreeNode`.

Core Note

While this section is concerned with tree nodes as part of the `JTree` component, it is worth bearing in mind that the tree model can be used on its own without reference to the tree control. You could, for example, use `DefaultTreeModel` to implement a tree to hold information used internally by your application. If you intend to do this and memory is a concern, you might consider implementing a lightweight implementation of `MutableTreeNode` that suits your specific requirements.

`DefaultMutableTreeNode` has too many methods to list them all here—refer to the API specification if you want to see what they all are. Many of them will be covered here and in the following sections.

The DefaultMutableTreeNode Class

The first thing we're going to do with this class is to show you how to use it to build a simple tree from scratch. `DefaultMutableTreeNode` has three constructors:

```
public DefaultMutableTreeNode()
public DefaultMutableTreeNode(Object userObject)
public DefaultMutableTreeNode(Object userObject,
                   boolean allowsChildren)
```

The first constructor creates a node with no associated user object; you can associate one with the node later using the `setUserObject` method. The other two connect the node to the user object that you supply. The second constructor creates a node to which you can attach children, while the third can be used to specify that child nodes cannot be attached by supplying the third argument as `false`.

Using `DefaultMutableTreeNode`, you can create nodes for the root and for all of the data you want to represent in the tree, but how do you link them together? You could use the `insert` method that we saw above, but it is simpler to use the `DefaultMutableTreeNode add` method:

```
public void add(MutableTreeNode child);
```

This method adds the given node as a child of the node against which it is invoked and at the end of the parent's list of children. By using this method, you avoid having to keep track of how many children the parent has. This method, together with the constructors, gives us all you need to create a workable tree. To begin to create a tree, you need a root node:

```
DefaultMutableTreeNode rootNode
                = new DefaultMutableTreeNode();
```

Below the root node, two more nodes are going to be added, one to hold details of the Apollo lunar flights, the other with information on the manned Skylab missions. These two nodes will be given meaningful text labels:

```
DefaultMutableTreeNode apolloNode =
                   new DefaultMutableTreeNode("Apollo");
DefaultMutableTreeNode skylabNode =
                   new DefaultMutableTreeNode("Skylab");
```

The nodes are then added directly beneath the root node:

```
rootNode.add(apolloNode);
rootNode.add(skylabNode);
```

Under each of these nodes, a further node will be added for each mission and beneath each of these a leaf node for each crew member. There's an implementation of this in the example programs that you can run using the command:

```
java JFCBook.Chapter10.TreeExample1
```

The result of running this example is shown in Figure 10-8.

Figure 10-8 A tree built using DefaultMutableTreeNodes.

This program shows a root folder with no associated label and nodes labeled `Apollo` and `Skylab`. Clicking on the expansion icons of either of these opens it to show the numbered missions, and clicking on any of these shows the crew for that flight. Let's look at an extract from the source of this example:

```
import javax.swing.*;
import javax.swing.tree.*;

public class TreeExample1 extends JTree {
  public TreeExample1() {
    DefaultMutableTreeNode rootNode =
                new DefaultMutableTreeNode();
    DefaultMutableTreeNode apolloNode =
              new DefaultMutableTreeNode("Apollo");
    rootNode.add(apolloNode);

    DefaultMutableTreeNode skylabNode =
              new DefaultMutableTreeNode("Skylab");
    rootNode.add(skylabNode);

    // CODE OMITTED

    this.setModel(new DefaultTreeModel(rootNode));
  }
  public static void main(String[] args) {
    JFrame f = new JFrame("Tree Example 1");

    TreeExample1 t = new TreeExample1();
    t.putClientProperty("JTree.lineStyle", "Angled");
    t.expandRow(0);

    f.getContentPane().add(new JScrollPane(t));
    f.setSize(300, 300);
    f.setVisible(true);
  }
}
```

This class is defined as an extension of `JTree`, which allows the creation of its data to be encapsulated within it. The root node and all of the child nodes are created and a tree structure is built from the nodes as described earlier. The `JTree` needs a data model in order to display anything, so the last step of the constructor is to install a model that contains the structure that has just been created:

```
this.setModel(new DefaultTreeModel(rootNode));
```

This creates a new `DefaultTreeModel` and initializes it with our root node, then uses the `JTree setModel` method to associate the data model with the tree. Since our class is derived from `JTree`, its default constructor will have been invoked at the start of our constructor. As noted earlier, this creates a tree with a model containing dummy data. When `setModel` is called at the end of the constructor, this data is overwritten with the real data.

Another way to create a `JTree` is to directly pass it the root node. If you use this method, it creates a `DefaultTreeModel` of its own and wraps it around the node that you pass to its constructor. Here's a short example of that:

```
DefaultMutableTreeNode rootNode =
                  new DefaultMutableTreeNode();
DefaultMutableTreeNode apolloNode =
                  new DefaultMutableTreeNode("Apollo");
DefaultMutableTreeNode skylabNode =
```

```
                    new DefaultMutableTreeNode("Skylab");
rootNode.add(apolloNode);
rootnode.add(skylabNode);
JTree t = new JTree(rootNode);
```

If you look at the `main` method in the code extract shown above, you'll notice the following line after the tree was created:

```
t.expandRow(0);
```

This line uses the `expandRow` method to ensure that row 0 of the tree is expanded to display the children that the node on that row contains. In fact, this line is redundant in this example because the root node is expanded by default. You can force a node to be shown in an unexpanded state by calling the `collapseRow` method, which also requires the index of the row within the tree. We'll say more about the methods that can be used to expand and collapse parts of the tree later in this chapter.

Apart from when you create it, the `JTree` control doesn't deal with nodes directly. Instead, you can address items in the tree and obtain information about them using either their `TreePath` or their row number. Let's look at the row number first. The row number refers to the number of the row on the screen at which the node in question appears. There is only one node ever on any row, so specifying the row identifies a node without any ambiguity. Furthermore, provided it's actually displayed, row 0 is always occupied by the root node. The problem with using row numbers is that the row numbers for all of the nodes apart from the root node change as nodes are opened or closed. When you start `TreeExample1`, the root node is on row 0, the "Apollo" node on row 1, and the "Skylab" node occupies row 2. However, if you click on the expansion icon for "Apollo," the "Skylab" node moves downward and, in this case, becomes row number 9, because the "Apollo" node opens to show seven child nodes, which will occupy rows 2 through 8. Because keeping track of row numbers is not very convenient, it is more usual to address the content of a tree using `TreePath` objects.

The TreePath Class

If `DefaultMutableTreeNode` is the physical representation of a node, then `TreePath` is its logical representation. Instead of pointing you directly at the node, it tells you the node's "address" within the tree. So long as the node itself and its parents are not moved within the tree, its `TreePath` will remain constant and can always be used to address it. `TreePath` has two public constructors:

```
public TreePath(Object singlePath) public TreePath(Object[] paths)
```

A `TreePath` stores an array of objects that describes the path to a node. The second constructor sets up this array from its argument, while the first is a convenience method that creates an array of length 1 and initializes it with the single object passed as the argument. `TreePath` has the following methods:

```
public boolean equals(Object)
public Object getLastPathComponent();
public Object getPathComponent(int index);
public int getPathCount();
public Object[] getPath();
public TreePath getParentPath();
public TreePath pathByAddingChild(Object child);
public boolean isDescendant(TreePath treePath)
public String toString()
```

None of these methods are concerned with what the objects that the `TreePath` stores actually are—they just treat them as anonymous objects. The `equals` method returns `true` if the object passed as its argument is another `TreePath` with an object array of the same length as its own and for which the result of comparing each object in its own array with the corresponding object in the other array (using the `equals` method the objects being compared) returns `true`.

The `getPath` method returns the array of objects installed by the constructor, while `getLastPathComponent` returns the last object in the array. In the case of a `TreePath` representing a set of directories from a file system tree ending in a file, this method would return the object that corresponds to the file itself. Similarly, `getPathCount` returns the number of objects in the path and `getPathComponent` returns a single component of that path given its index, where 0 is the component at the root. `getParentPath` returns a `TreePath` object representing the parent of the `TreePath` against which it is invoked. `pathByAddingChild` creates and returns a `TreePath` object that represents the results of appending the `Object` passed as its argument to the current `TreePath`. In terms of the file system tree, if you invoked this method against the `TreePath` for a

node representing directory and passed it an `Object` representing a file or subdirectory within that directory, it would return a `TreePath` for that file or subdirectory.

The `isDescendant` method determines whether the `TreePath` passed as its argument represents a descendant of the `TreePath` against which it is invoked. This will be true if the `TreePath` passed as the argument has at least as many objects in its array as the `TreePath` against which this method is invoked, and each of the objects in the invoked `TreePath`'s array is equal to the corresponding object in the array of the proposed descendant. Finally, the `toString` method prints a readable representation of the `TreePath` by invoking the `toString` methods of each object in its array and concatenating them, separated by commas.

The `TreePath` doesn't care what the objects that it manipulates are, but you need to know what they are in order to make much use of a `TreePath`. The objects that `TreePath` actually stores are the `TreeNode`s for each node of the tree between the root and the object that the `TreePath` relates to. In most cases (and in all of the examples in this chapter), this object will actually be a `MutableTreeNode`. This means that, in an event handler that receives a `TreePath` in its event, you can access the information from the data model that the event relates to, including the user object associated with the node, simply by getting a reference to the node from the `TreePath`. You'll see when looking at the file system tree component that is developed later in this chapter why this is a useful feature.

Let's look in more detail at what a `TreePath` is made up of by using an example. `TreeExample2` is a development of the last example in which the tree is built and then, every 30 seconds, the `TreePath` object for whatever is on row 4 of the tree's display is obtained and its content is printed. Here is the interesting part of this program:

```
for (;;) {
  Thread.sleep(30000);

  // Get TreePath for row 4
  TreePath p = t.getPathForRow(4);
  if (p == null) {
    System.out.println("Nothing on row 4!");
    continue;
  }
  // Print the official description of "p"
  System.out.println("==================\n" + p);

  // Now look inside
  Object[] o = p.getPath();
  for (int i = 0; i < o.length; i++) {
    System.out.println("Class: " + o[i].getClass() +
       "; value: " + o[i]);
    if (o[i] instanceof DefaultMutableTreeNode) {
     Object uo =
          ((DefaultMutableTreeNode)o[i]).getUserObject();
     if (uo != null) {
       System.out.println("\tUser object class: " +
         uo.getClass() +"; value: " + uo);
     }
    }
  }
}
```

The `TreePath` for whatever is on row 4 is obtained by using the `get-PathForRow` method of `JTree`. If the row is empty, as it will be when you start the program, this method returns `null`, so a message is printed and nothing further happens for another 30 seconds. Otherwise, the `TreePath toString` method is used to see its own description of itself, then `getPath` is called to get its array of component parts. For each object returned in this array, its Java class and its value are printed. Finally, if any object that is obtained from the array is a `DefaultMutableTreeNode`, its `getUserOb-ject` method is called to get the user object associated with the node and, if there is one, its class and its value are also printed. You can run this example yourself by typing:

```
java JFCBook.Chapter10.TreeExample2
```

Following are the results of three iterations around the loop with three different directories expanded to place different objects on row 4 of the display.

```
 [null, Skylab, 3]
Class: class javax.swing.tree.DefaultMutableTreeNode; value: null
Class: class javax.swing.tree.DefaultMutableTreeNode; value: Skylab
      User object class: class java.lang.String; value: Sky-
Lab
```

```
Class: class javax.swing.tree.DefaultMutableTreeNode; value: 3
       User object class: class java.lang.String; value: 3
 [null, Apollo, 13]
Class: class javax.swing.tree.DefaultMutableTreeNode; value: null
Class: class javax.swing.tree.DefaultMutableTreeNode; value: Apollo
       User object class: class java.lang.String; value:
Apollo
Class: class javax.swing.tree.DefaultMutableTreeNode; value: 13
       User object class: class java.lang.String; value: 13
 [null, Apollo, 12, Pete Conrad]
Class: class javax.swing.tree.DefaultMutableTreeNode; value: null
Class: class javax.swing.tree.DefaultMutableTreeNode; value: Apollo
       User object class: class java.lang.String; value:
Apollo
Class: class javax.swing.tree.DefaultMutableTreeNode; value: 12
       User object class: class java.lang.String; value: 12
Class: class javax.swing.tree.DefaultMutableTreeNode; value: Pete Conrad
       User object class: class java.lang.String; value: Pete
Conrad
```

The `toString` method of `TreePath` prints the path to the node that was on row 4. Next, the objects that correspond to the various parts of the path are shown. As you can see, there is one object for each node in the path, starting at the root and ending with the node that corresponds to the `Tree-Path`. As noted above, these objects are actually the nodes that were stored in the `DefaultTreeModel`, so they are all instances of `DefaultMutable-TreeNode` and invoking `toString` on them prints the string that was supplied to the constructor that created them. Finally, each of these nodes, apart from the root node, has an associated user object that, as you can see, is the same string. This is not, however, a surprise, because the `DefaultMutable-TreeNode` constructor that was used described its argument as the user object associated with the node. In fact, the `toString` method of `Default-MutableTreeNode` just invokes the `toString` method of the user object if there is one and returns an empty string if there isn't. This explains why the `DefaultMutableTreeNode` returns the string passed to its constructor.

Incidentally, we now have the information needed to modify one of our earlier example programs so that the string `root` doesn't appear next to the root node on the display, as it did in Figure 10-7. In that example, the root nodes were not explicitly created by the program: Because each `JTree` was built by passing a `Hashtable` to its constructor, the root nodes were built for us. The string displayed for the root node is the result of invoking `toString` against that node. You now know that, because this node is a `DefaultMu-tableTreeNode`, its `toString` method will delegate to that of its user object if there is one and return an empty string otherwise. When the root nodes in Figure 10-7 were created, the string `root` was assigned as the user object, so to make it disappear, you just have to use the `setUserObject` method to set the user object to `null`.

Here is how you would create a tree that shows a root folder without the `root` label:

```
JTree t = new JTree(h);
t.setShowsRootHandles(false);
t.setRootVisible(true);

// Remove the 'root' label
Object rootNode = t.getModel().getRoot();
((DefaultMutableTreeNode)rootNode).setUserObject(null);
```

This code gets a reference to the model from the `JTree` using the `get-Model` method and then gets the root node from the model by using the `TreeModel getRoot` method. This method is defined to return an `Object`, but you know that the root node is a `DefaultMutableTreeNode`, so all you have to do is cast it and use `setUserObject` to remove the `root` string. You can use the same technique to assign an arbitrary label to the root node. You can see how this works in practice using the command

```
java JFCBook.Chapter10.HashHandleTree3
```

which produces the same result as that shown in Figure 10-7, except that the root nodes do not have any associated text.

What is a Leaf?

The Metal look-and-feel displays nodes that it considers to be leaves using what looks like a sheet of paper instead of a folder icon. How does it determine that the node is a leaf? There are two methods in the `TreeNode` interface that determine whether a node is a leaf:

```
public boolean getAllowsChildren()
public boolean isLeaf()
```

If you think of the tree of Apollo and Skylab missions and their astronauts, the astronauts are represented as leaf objects. This is correct, because it is not our intention to add further nodes below those representing the astronauts. In cases like this, it is acceptable to determine that a node is a leaf simply because it doesn't actually have any children. Contrast this, however, with a file system. If a file system were represented by a tree control (an example of which you'll see later), you would expect files to be leaf objects, but you would always expect a directory to be represented as a folder, whether or not it contain any files or subdirectories. In this case, it is not sufficient to rely on the number of children that the node has.

If relying on the count of children is acceptable to determine whether a node is a leaf, the `isLeaf` method returns the appropriate answer, because the default implementation in `DefaultMutableTreeNode` returns `true` if the node that it is applied to does not have children and `false` if it does. This explains why the astronauts all appeared as leaf nodes even though nothing special was done when creating their nodes.

For nodes like directories, a more appropriate check is not whether the node has any child nodes, but whether it is *allowed* to have child nodes attached to it. If it isn't, then it must be a leaf and vice versa. The `getAl-lowsChildren` method is used to allow the node to return this information. `DefaultMutableTreeNode` supplies a default implementation that returns the value of the node's `allowsChildren` variable, which is initially `true`, indicating that the node is not a leaf.

The next problem, given that there are two ways to find out whether a node is a leaf, is how the tree knows which method is the appropriate one to use. This problem is handled by `DefaultTreeModel`, which maintains a boolean instance variable called `asksAllowsChildren`, initialized to `false` by default. `DefaultTreeModel` has two constructors:

```
public DefaultTreeModel(TreeNode rootNode);
public DefaultTreeModel(TreeNode rootNode,
                        boolean asksAllowsChildren)
```

If the second constructor is used, you can supply `true` as the second argument to indicate that all nodes in the model should have their `getAl-lowsChildren` method called to determine whether they should be considered a leaf. If `asksAllowsChildren` is `false`, the node's `isLeaf` method is always used. To achieve the correct effect, you need to use the correct tree model constructor and set this variable appropriately.

The value of this variable is only used in the `DefaultTreeModel isLeaf` routine:

```
public boolean isLeaf(Object node);
```

This method checks the `asksAllowsChildren` variable and returns the value of either `isLeaf` or `getAllowsChildren`, having invoked it on the `TreeNode` given as the argument. This method, then, returns the correct value whichever way the tree nodes are used to indicate whether they are leaf nodes.

Core Alert

Despite appearances, the argument supplied to this method must be a TreeNode*, or you'll get a* ClassCastException*.*

If you are using the `getAllowsChildren` method to determine whether a node is a leaf, you need to set the node's `allowsChildren` value properly when you create it. Alternatively, you can use the `setAllowsChildren` method to change it after construction.

Expanding and Collapsing a Tree

Even if you never change the content of a tree after creating it, trees are not static objects. Once it has been displayed, the user can change a tree's appearance by expanding and collapsing nonleaf nodes to show or hide different levels of the tree. This section shows you what possibilities there are for expanding and traversing a tree and looks at the events that these actions generate.

Tree Expansion and Traversal

When a tree is first created, unless you take special action, one level of nodes is visible. Nodes that have

children are rendered with an expansion handle, which you can use to make their child nodes visible. When you click on the handle, the next level of nodes is drawn and the icon changes its appearance to indicate that a second click will reverse the process and collapse the node. Another way to expand or collapse a subtree is to double-click on the node itself instead of clicking the expansion box. In Java 2 version 1.3, you can change the number of clicks that are needed to expand or collapse a node by clicking on the node itself:

```
public void setToggleClickCount(int clickCount);
```

It is also possible to navigate the tree using the keyboard. To see how this works, run the `TreeExample1` program that was shown earlier:

```
java JFCBook.Chapter10.TreeExample1
```

When this starts, you'll see the root node and two child nodes labeled `Apollo` and `Skylab`. Provided you don't click the mouse, none of these items will be highlighted. Now press the right-arrow key on your keyboard and the root node will be highlighted, indicating that it has been selected. Pressing the down arrow now moves the selection to the `Apollo` node and repeating it carries the selection down to `Skylab`. Similarly, the up arrow key moves the selection back up.

Now use the up and down arrow keys to highlight `Apollo` and then press the right arrow. This causes the `Apollo` node to expand and display its child nodes. With these nodes exposed, the down key now moves you not to `Sky-lab`, but to `11`, the first child node of `Apollo`. In fact, the up and down keys just move the selection up and down by one row on the screen. If you leave the selection on `11` and press the right arrow key, this node opens to expose the names of the crew of Apollo 11. With these nodes visible, press the left arrow key. This collapses the subtree rooted at `11`, but leaves `11` selected. So it appears that the left and right arrows just expand and collapse a node's subtree. However, if you press the right arrow with the selection on `11` to expand its child nodes, then press it again, the selection moves to the first child. In other words, the left and right arrow keys open and close a subtree if they can, but if the subtree is already open or closed, these keys behave like the up and down arrows, respectively.

You can also use the `HOME`, `END`, `PAGE DOWN` and `PAGE UP` keys to navigate a tree. The `HOME` key moves the selection back to the root, or whichever node is at the top of the tree if the root node is hidden, while `END` moves it to the last row of the tree. `PAGE DOWN` and `PAGE UP` scroll the tree down or up by a page. To see this, expand the tree completely and, if necessary, resize the window until the vertical scrollbar appears so that you have more in the view-port than can be displayed on one page, then press `PAGE DOWN`.

Expanding and Collapsing Nodes under Program Control

`JTree` provides a set of methods that you can use to expand or collapse parts of the tree and to find out whether a given node is expanded or not. These methods are listed in Table 10-2.

Table 10-2 Expanding and Collapsing Nodes in a Tree

Method	Description
`public void expandRow (int row)`	Expands the node currently on the given row of the screen.
`public void collapseRow(int row)`	Collapses the node on the given row.
`public void expandPath(TreePath path)`	Expands the node with the given `TreePath`.

`public void collapsePath(TreePath path)`	Collapses the node with the given `TreePath`.
`public boolean isExpanded(int row)`	Determines whether the node on the given row is expanded.
`public boolean isCollapsed (int row)`	Determines whether the node on the given row is collapsed.
`public boolean isExpanded(TreePath path)`	Determines whether the node with the given `TreePath`is expanded.
`public boolean isCollapsed (TreePath path)`	Determines whether the node with the given `TreePath`is collapsed.

The `isExpanded` and `isCollapsed` methods just return a `boolean` that indicates whether the node is currently expanded or collapsed. You can address the node either using its screen row (which is rarely likely to be a useful option), or its `TreePath`, which is more useful because it is independent of how much of the tree is expanded at any given time. Notice, however, that there is no method that can be used with a `TreeNode`, because these are `JTree` methods and `JTree` doesn't deal with `TreeNodes`, except in its constructor. However, if you have a `DefaultMutableTreeNode`, you can get the corresponding `TreePath` using the following code:

```
// This only works if "node" is a DefaultMutableTreeNode.
DefaultMutableTreeNode node = ..... // Initialize to point
                                    // to a node in the tree

TreeNode[] pathToRoot = node.getPath();
TreePath path = new TreePath(pathToRoot);
```

Given a node, the `DefaultMutableTreeNode getPath` method returns the ordered set of nodes that lead to it from the root of the tree. For example, in the case of `TreeExample1`, the ordered set of nodes that would be returned by `getPath` for the node corresponding to `Neil Armstrong` would be:

```
root
Apollo
11
Neil Armstrong
```

Using a set of nodes, you can create a `TreePath` by using the constructor that takes an array of objects—as you saw earlier, the objects that make up a `TreePath` are actually just the `TreeNodes` in the path that it represents. This is a general method for mapping from a node to a `TreePath`, but it only works with a `DefaultMutableTreeNode` because neither the `TreeNode` nor the `MutableTreeNode` interface requires the implementation of the `getPath` method that was used to get the array of nodes.

The `collapseRow` and `collapsePath` methods perform the obvious functions. Again, you must know either a row number or a `TreePath` to use these methods. Nothing happens if the node is already collapsed.

To expand a node, you use either `expandRow` or `expandPath`. If the node has any children and it is not already expanded, its immediate children are displayed. If you want to expand more of the tree, you will need to apply the expand methods to subsequent levels. If the node that is the target of these methods is not itself visible when asked to expand, the parts of the tree that lead to it will first be expanded to make it visible. For example, if the example tree were completely closed and you asked for the node `11` to be expanded, the root node would be expanded to show `Apollo` and `Skylab`, then `Apollo` would be expanded to show all of the mission numbers, including `11`, and finally `11` would expand to show `Neil Armstrong`, `Buzz Aldrin`, and `Michael Collins`. In this case, expanding a node and then collapsing it does not leave the tree in the same state as it

started, because the nodes above `11` would not be collapsed. Note, however, that you can only expand or collapse a child that is not a leaf node. Therefore, if you want to expand the tree to show the node for `Neil Armstrong`, you would have to call `expandPath` on the parent of that node, not on the node itself. When a node is expanded, the tree is automatically scrolled to bring as many as possible of the node's children into view (assuming, of course, that it is mounted in a `JScrollPane`). If you don't want this behavior, you can disable it using the first of the following `JTree` methods:

```
public void setScrollsOnExpand(boolean cond);
public boolean getScrollsOnExpand();
```

Tree Expansion Events

Every time a node in the tree is expanded or collapsed, a `TreeExpansion-Event` is generated. You can receive these events by registering a `TreeEx-pansionListener` on the `JTree` itself. The `TreeExpansionListener` interface has only two methods:

```
public void treeExpanded(TreeExpansionEvent event)
public void treeCollapsed(TreeExpansionEvent event)
```

The `TreeExpansionEvent` contains a source field that corresponds to the `JTree` itself and a `TreePath` that represents the node was expanded or collapsed. You can obtain the source of the events using the usual `get-Source` method and the path using `getPath`:

```
public TreePath getPath();
```

Sometimes, expanding a path can generate more than one event, because it may be necessary to expand other nodes to make the target node visible. To see the events that are generated when this happens, we created a modified version of `TreeExample1` that registers a `TreeExpansionListener` and prints the events that it receives. Here is the snippet of code that was added to implement the listener:

```
t.addTreeExpansionListener(new TreeExpansionListener() {
 public void treeExpanded(TreeExpansionEvent evt) {
   System.out.println("Expansion event, path " +
                      evt.getPath());
 }

 public void treeCollapsed(TreeExpansionEvent evt) {
   System.out.println("Collapse event, path " +
                      evt.getPath());
 }
});
```

You can run the modified version of the program using the command:

```
javac JFCBook.Chapter10.TreeExample3
```

The code was also changed so that the tree starts with only the root visible by adding the line:

```
t.collapseRow(0);
```

If you expand and collapse nodes in the tree, you'll see the events that are generated displayed in the window from which you started the program. Notice that there is one event generated each time you expand or collapse a node. This remains true even if you open the whole tree and then collapse it by double-clicking on the root node. The event that you get contains a `Tree-Path` for the root: You don't get events for all of the other nodes that disappeared. This is actually reasonable, because if you expand the root again, you'll find that the tree returns to its fully expanded state—in other words, the other nodes didn't actually collapse. As well as adding a listener, code was also added to expand the node corresponding to `11`, which runs after a 15-second delay. Here is the extra code for this:

```
Timer timer = new Timer(15000, new ActionListener() {
  public void actionPerformed(ActionEvent evt) {
    DefaultMutableTreeNode rootNode =
       (DefaultMutableTreeNode)t.getModel().getRoot();
    DefaultMutableTreeNode apolloNode =
       (DefaultMutableTreeNode)rootNode.getFirstChild();
    DefaultMutableTreeNode apollo11Node =
    (DefaultMutableTreeNode)apolloNode.getFirstChild();

    // Convert the node to a TreePath
    TreeNode[] pathToRoot = apollo11Node.getPath();
```

```
    TreePath path = new TreePath(pathToRoot);

     // Expand the node
     t.expandPath(path);
  }
});
timer.setRepeats(false);
timer.start();
```

The first problem is to find the node that needs to be opened. This is achieved by getting the root node from the tree's model and then using the `getFirstChild` method of `DefaultMutableTreeNode` to walk down through the various levels until the target is reached. `DefaultMutable- TreeNode` has several methods that allow you to access related nodes—look at its API documentation to find out what they are.

Having found the target node, it must be converted to a `TreePath`, for which the technique shown earlier is used, then the node can finally be expanded. If you start the program and do nothing for 15 seconds, you'll find that the tree opens to show the crew of Apollo 11, and three events are generated, one for the target node and one for each level above the target that was expanded automatically to make the target node visible:

```
Expansion event, path [null]
Expansion event, path [null, Apollo]
Expansion event, path [null, Apollo, 11]
```

Making Nodes Visible

When you change the state of a tree programmatically, it is sometimes useful to have the tree scroll so that the user can see the affected area. In the example that you have just seen, the frame was large enough so that the whole tree was visible after it was expanded to show the node for `Neil Armstrong`. However, if you run the example again and make the window so short that you can only see the root node, when the tree expands you won't be able to see what has happened.

If you want to force a particular node to be visible, there are two `JTree` methods that you can use:

```
public void scrollPathToVisible(TreePath path);
public void scrollRowToVisible(int row);
```

Naturally, these methods only work if the tree is mounted in a `JScroll-Pane` and they have no effect if the node specified is already within the visible area of the `JScrollPane`. The following command runs an improved version of the previous example that forces the node for `Neil Armstrong` into view once its parent node has been expanded:

```
java JFCBook.Chapter10.TreeExample4
```

The code to do this was added after the `expandPath` call and looks like this:

```
// Make Neil Armstrong node visible
DefaultMutableTreeNode targetNode =
   (DefaultMutableTreeNode)apollo11Node.getFirstChild();
path = path.pathByAddingChild(targetNode);
t.scrollPathToVisible(path);
```

You'll notice that, since we already had a `TreePath` object for the parent of the node that we wanted to reach, we were able to use the `TreePath pathByAddingChild` method to get a `TreePath` for the `Neil Armstrong` node, once we obtained its `DefaultMutableTreeNode`. To demonstrate that this works, when the example starts, resize the window so that it is tall enough to show only the root node and then wait for the tree to expand itself. When it does so, you'll see that it also scrolls to bring `Neil Armstrong` into view. Incidentally, the `scrollPathToVisible` method can cause both horizontal and vertical scrolling, so if the window were also very narrow, it would scroll horizontally to bring the node into view if necessary.

Controlling Node Expansion and Collapse

As you have seen, you can receive notification that a tree node has been expanded or collapsed by registering a `TreeExpansionListener`. Sometimes, it is useful to be notified that a node is going to be expanded or collapsed before the change is actually made. `JTree` supports this by first sending the `TreeExpansionEvent` for the expansion or collapse to any registered `TreeWillExpandListener`s. `TreeWillExpandListener` is an interface with two methods:

```
public interface TreeWillExpandListener
            extends EventListener {
  public void treeWillCollapse(TreeExpansionEvent evt)
                throws ExpandVetoException;
  public void treeWillExpand(TreeExpansionEvent evt)
                throws ExpandVetoException;
}
```

This interface is almost the same as `TreeExpansionListener`, apart from the fact that the listener methods can throw an `ExpandVetoException`. Before any node is expanded or collapsed, the tree creates the appropriate `TreeExpansionEvent` and delivers it to each registered `TreeWillEx-pandListener`. If every listener returns without throwing an exception, the node changes state and the `TreeExpansionEvent` is then delivered to all `TreeExpansionListeners`. However, if any listener throws an `ExpandVe-toException`, the node will not be expanded or collapsed and no `TreeEx-pansionListeners` will receive the `TreeExpansionEvent`.

You can use a `TreeWillExpandListener` to decide whether to allow part of a tree to be expanded or collapsed based on any criterion you chose. Suppose, for example, we want to change our ongoing tree example so that, once the Apollo 11 node has been made visible, the tree cannot be collapsed in such a way as to hide that node. All we need to do to enforce this is to add a `TreeWillExpandListener` and perform the appropriate checks in the `treeWillCollapse` method. You can run a version of the example that has this feature using the command

```
java JFCBook.Chapter10.TreeExample5
```

When this example starts, if you wait for 15 seconds, the tree will open to show the node for `Neil Armstrong` and you'll see the following events delivered:

```
Tree will expand event, path [null]
Expansion event, path [null]
Tree will expand event, path [null, Apollo]
Expansion event, path [null, Apollo]
Tree will expand event, path [null, Apollo, 11]
Expansion event, path [null, Apollo, 11]
```

As you can see, even though the code only requests that one node be expanded, because its parent nodes also need to be opened in order to make the target node visible, the `TreeWillExpandListener treeWillExpand` method is invoked before each node is expanded. Since no `ExpandVetoEx-ception`s were thrown, the operation completes normally.

Now click on the `11` node. Under normal circumstances, this would cause the node to close, but on this occassion, it does not and the following output appears in the command window:

```
Tree will collapse event, path [null, Apollo, 11]
Veto collapse of path
```

In this case, the collapse has been vetoed, so the node stays in its expanded state. The same thing happens if you click on either the `Apollo` node or the root node, because these are all the nodes that could be closed to hide the node for `Neil Armstrong`. On the other hand, you can still open and close the other `Apollo` nodes and all of the `Skylab` nodes.

Here is the code that has been added to enforce this policy:

```
t.addTreeWillExpandListener(new TreeWillExpandListener()
{
  public void treeWillExpand(TreeExpansionEvent evt)
         throws ExpandVetoException {
    System.out.println("Tree will expand event, path " +
                    evt.getPath());
  }

  public void treeWillCollapse(TreeExpansionEvent evt)
         throws ExpandVetoException {
    System.out.println("Tree will collapse event, path "
                        + evt.getPath());
    DefaultMutableTreeNode rootNode =
       (DefaultMutableTreeNode)t.getModel().getRoot();
    DefaultMutableTreeNode apolloNode =
       (DefaultMutableTreeNode)rootNode.getFirstChild();
    DefaultMutableTreeNode apollo11Node =
       (DefaultMutableTreeNode)apolloNode.getFirstChild();
    // Build the path to the Apollo 11 node
    TreeNode[] pathToRoot = apollo11Node.getPath();
```

```
    TreePath apollo11Path = new TreePath(pathToRoot);

    // Don't allow Apollo 11 node or any ancestor
    // of it to collapse.
    if (evt.getPath().isDescendant(apollo11Path)) {
      System.out.println("Veto collapse of path");
      throw new ExpandVetoException(evt);
    }
  }
}) ;
```

The `treeWillCollapse` method contains the code of interest. It first constructs a `TreePath` for the `11` node in the normal way. Having done this, it uses the `TreePath isDescendent` method to determine whether the `11` node is a descendent of the node whose `TreePath` is delivered in the `Tree-ExpansionEvent`. If it is, then the operation should not be permitted, so an `ExpandVetoException` is thrown. Notice that the `ExpandVetoException` constructor requires the `TreeExpansionEvent` that is being vetoed as an argument.

Selecting Items in a Tree

If you want a tree to be useful for user feedback, you need to be allow the user to select items and to take action when those selections are made. `JTree`, like `JList`, has a separate selection model that offers several different selection modes. This section looks at how to control and respond to a selection within a tree.

Selection and the TreeSelectionModel

`JTree` delegates management of the selection to a selection model. By default, an instance of the class `DefaultTreeSelectionModel` is associated with the tree when it is created, but you can, if you wish, define your own selection model, as long as it implements the `TreeSelectionModel` interface. You can switch in your custom selection model using the `setSe-lectionModel` method. In this section, it is assumed that the default selection model is in use.

The selection model can be configured to operate in one of three different modes:

- `SINGLE_TREE_SELECTION`: only one node from the tree can be selected at any given time.
- `CONTIGUOUS_TREE_SELECTION`: any number of nodes can be selected, as long as they occupy adjacent rows.
- `DISCONTIGUOUS_TREE_SELECTION`: any number of nodes can be selected, with no restriction on location relative to other nodes.

The default mode is DISCONTIGUOUS_TREE_SELECTION.

Changes in the selection can be made either under program control or by the user. No matter how the selection is changed, the selection model ensures that any changes fit in with the current selection mode. Changes to the selection also generates an event, as you'll see later.

The user can create or change the selection by using the mouse or the keyboard. Clicking on a node or pressing `SPACE` when the node has the focus will make the selection contain just that single node. Clicking or pressing `SPACE` on another node removes the first node from the selection and selects only the item just clicked on. Selection of a single item in this way is always possible, unless there is no selection model, in which case no selection is allowed. You can create this situation with the following lines of code:

```
JTree t = new JTree()
t.setSelectionModel(null);   // No selection allowed
```

Core Warning

If you disable all selection the user will be unable to navigate the tree and expand or collapse nodes using the keyboard. For this reason, it is usually better not to completely disallow selection, but instead to change the mode of the `TreeSelectionModel` to `SINGLE_TREE_SELECTION`. If you don't want the selected node to look selected, you can implement a custom renderer that will draw the selected node differently, perhaps by drawing a border around it but not filling the background with the selection color. Custom renderers are discussed later in this chapter.

Holding down the CTRL key while clicking on a node (or pressing the SPACE key) adds that node to the selection, leaving the existing content of the selection intact. If the node clicked on is already in the selection, it is removed. However, there are some restrictions on this, depending on the selection mode:

- If the selection mode is SINGLE_TREE_SELECTION, the CTRL key is ignored—that is, if there is already a node in the selection it is de-selected and the new node replaces it.
- If the selection mode is CONTIGUOUS_TREE_SELECTION, the new node must be adjacent to the row or rows that make up the current selection. If this is not the case, the selection is cleared and the new node becomes the only member of the selection.
- If the selection mode is CONTIGUOUS_TREE_SELECTION and the first or last row in the selection is clicked with CTRLpressed, that node is removed from the selection leaving the rest of the nodes selected.
- If the selection mode is CONTIGUOUS_TREE_SELECTIONand the node clicked is one of the selected nodes but is not at the top or bottom of the selected range, the selection is completely cleared. The node clicked is *not* added to the selection.

Holding down the SHIFT key while clicking or navigating the tree with the arrow keys creates a contiguous selection between two selected nodes. This is not allowed if the selection model is in SINGLE_TREE_SELECTION mode. In that case, the clicked node would become the selected node.

When a selection is made, the path most recently selected is often the one of most interest to the application. This path is called the *lead selection* path and its TreePath or row number can be specifically requested by the program using the getLeadSelectionPath or getLeadSelectionRow methods (see Table 10-3).

Core Note

There is one case in which the lead row is not the last row selected—see the discussion of tree selection events below for details.

There are, in fact, several distinguished rows and paths that the tree and/ or the selection model treat as special:

Minimum Selection Row	The lowest numbered row that is part of the selection. This is simply the row number of the selected node nearest the top of the tree.
Maximum Selection Row	The highest numbered row that is part of the sele-tion. Not surprisingly, this is the row number of the selected node nearest the bottom of the tree.
Lead Selection Row or Path	Usually the last node to be added to the selection.
Anchor Selection Path	When only one node is selected, this is the same as the lead selection. When a row is added to an existing selection using the CTRL key, that row becomes both the lead and anchor selection. When a selection already exists and it is extended by using the

	SHIFT key, the anchor selection is the row that was already selected and which forms the other end of the selection from the most recently selected row (which is the lead selection). The concept of the anchor selection is supported only by JTree and was introduced in Java 2 version 1.3

There is a collection of methods, listed in Table 10-3, that can be used to query or change the selection. Unless otherwise indicated, all of these methods are provided by both DefaultTreeSelectionModel and, for convenience, by JTree. There are also several methods that are only available from JTree.

The selection can span many rows of a tree; an extreme case would be opening the whole tree, clicking on the root node, then scrolling to the bottom of the tree and holding down SHIFT while clicking on the last node which, unless the tree is in SINGLE_TREE_SELECTION mode, would select every node in the tree. Keep in mind, however, that only nodes that are visible can be selected using the mouse: Selecting a node that has child nodes does not select any of the child nodes (and vice versa). Also, if you select a node or nodes in a subtree and then collapse that subtree so that those nodes are no longer visible, they become deselected and the selection passes to the root of the closed subtree, whether or not it was selected when the node was collapsed.

It is possible to programmatically select nodes that are not visible. What happens when you do this depends on the version of Swing that you are using. In Swing 1.1.1 and Java 2 version 1.2.2, when you select an invisible node, its

Table 10-3 Methods Used to Manage the Selection

Method	Description
public int getSelectionMode()	Returns the current tree selection mode.
public void setSelectionMode(int mode)	Sets the tree selection mode.
public Object getLastSelectedPathComponent()	If there is an item selected, returns its last path component, or null if there is no selected item. The result of calling this method when there is more than one item selected is not very useful because it does not always use the most recently selected item. This method is provided only by JTree;it is not supported by DefaultTreeSelection-Model.

public TreePath getAnchorSelectionPath() (Java 2 version 1.3 only)	Returns the path for the anchor selection. This method is provided only by JTree;it is not supported by DefaultTreeSelection-Model.
public voidsetAnchorSelectionPath(TreePathpath)	
(Java 2 version 1.3 only)	Sets the anchor selection path. This method is provided only by JTree; it is not supported by Default—TreeSelectionModel
public TreePath	Returns the path for the lead selec-
getLeadSelectionPath()	tion.
public void setLeadSelectionPath()	Sets the path for the lead selection.
(Java 2 version 1.3 only)	This method is provided only by JTree; it is not supported by DefaultTreeSelection-Model.
public int getMaxSelectionRow()	Returns the highest selected row number.
public int getMinSelectionRow()	Returns the lowest selected row number.
public int getSelectionCount()	Gets the number of selected items.
public TreePath getSelectionPath()	Returns the TreePathobject for the first item in the selection.
public TreePath[] getSelectionPaths()	Returns the TreePathobjects for all of the selected items.
public int[] getSelectionRows()	Returns the row numbers of all of the items in the selection.

`public boolean isPathSelected (TreePath path)`	Returns `true` if the given path is selected, `false` if not.
`public boolean isRowSelected(int row)`	Returns `true` if the given row is selected, `false` if not.
`public boolean isSelectionEmpty()`	Returns `true` if there are no selected items.
`public void clearSelection()`	Removes all items from the selection.
`public void removeSelectionInterval (int row0, int row1)`	Removes the rows in the given range from the selection. This method is provided only by `JTree`-it is not supported by `DefaultTreeSelectionModel`.
`public void removeSelectionPath(TreePath path)`	Removes one path from the selection.
`public void removeSelectionPaths(TreePath[] paths)`	Removes the listed paths from the selection.
`public void removeSelectionRow (int row)`	Removes the given row from the selection. This method is provided only by `JTree`;it is not supported by `DefaultTreeSelection-Model`.
`public void removeSelectionRows(int[] rows)`	Removes the listed rows from the selection. This method is provided only by `JTree`;it is not supported by `DefaultTreeSelection-Model`.
`public void addSelectionInterval(int row0, int row1)`	Adds the rows in the given range to the current selection.
`public void addSelectionPath(TreePath path)`	Adds the given path to the selection.
`public void addSelectionPaths(TreePath[] path)`	Adds the given set of paths to the selection.

`public void addSelectionRow(int row)`	Adds the object on the given row to the selection.
`public void addSelectionRows(int[] row)`	Adds the objects in the given rows to the selection.
`public void setSelectionInterval(int row0, int row1)`	Makes the selection equal to all rows in the given range. This method is provided only by `JTree`; it is not supported by `DefaultTreeSelectionModel`.
`public void setSelectionPath(TreePath path)`	Makes the selection be the given path.
`public void setSelectionPaths (TreePath[] path)`	Sets the selection to the given set of paths.
`public void setSelectionRow(int row)`	Sets the selection to the object on the given row. This method is provided only by `JTree`; it is not supported by `DefaultTreeSelection-Model`.
`public void setSelectionRows(int[] row)` Sets the selection to the set of objects in the given rows. This method is provided only by `JTree`; it is not supported by `Default`	`TreeSelectionModel`.

parent and any other collapsed descendents are expanded so that the newly selected node is visible. In Java 2 version 1.3, this is also the default behavior, but you can call the following method with argument `false` to disable it:

```
public void setExpandsSelectedPaths(boolean cond);
```

Tree Selection Events

A change in the selection is reported using a `TreeSelectionEvent`. To receive these events, you must register a `TreeSelectionListener` using the `addTreeSelectionListener` method of your tree's `TreeSelection-Model`, or the convenience method of the same name provided by `JTree`. The `TreeSelectionListener` interface has only one method:

```
public void valueChanged(TreeSelectionEvent evt)
```

You need to inspect the accompanying event to determine what changes were made to the selection.

`TreeSelectionEvent`s tell you exactly which paths are involved in the last change of selection and whether they

were added or removed from the selection. You can also get the leading path and the old leading path from the event. The important methods supplied by `TreeSelectionEvent` are shown in Table 10-4.

Table 10-4 Methods Used with Tree Selection Events

Method	Description
`public TreePath[] getPaths()`	Returns the paths for all of the nodes that were affected by the last selection change.
`public boolean isAddedPath (TreePath path)`	Assuming that `path` represents a node affected by this event, this method returns `true` if the node was added to the selection and `false` if it was removed from the selection.
`public boolean isAddedPath (int index)`	Returns `true` if the path in entry `index` of the paths delivered by this event was added to the selection, `false` if it was removed from the selection. This method was added in Java 2 version 1.3
`public TreePath getPath()`	Returns the first `TreePath` in the set that would be returned by `getPaths`. This is a convenience method for the case where you know there is only one path of interest.
`public boolean isAddedPath()`	Returns `true`if the path returned by `getPath`was added to the selection and `false`if it was removed.
`public TreePath getOldLeadSelection()`	Returns that path for the node that was the lead selection before this event.
`public TreePath` getNewLeadSelection()	Gets the path for the node that is now the lead selection.

The simplest way to understand how `TreeSelectionEvent`s work is to see an example. The program `JFCBook.Chapter10.TreeSelection-Events` uses the tree that has been used throughout this chapter, together

with some code to receive and print the contents of selection event:

```
t.addTreeSelectionListener(new TreeSelectionListener() {
  public void valueChanged(TreeSelectionEvent evt) {
    System.out.println("=======\nFirst path: " +
                       evt.getPath()
          + "; added? " + evt.isAddedPath());
    System.out.println("Lead path: " +
                       evt.getNewLeadSelectionPath());
    System.out.println("Old Lead path: " +
                       evt.getOldLeadSelectionPath());

    TreePath[] paths = evt.getPaths();
    for (int i = 0 ; i < paths.length; i++) {
      System.out.println("Path: < " + paths[i] + "; added? " +
        evt.isAddedPath(paths[i]));
      }
      System.out.println("Tree's lead selection path is " +
                       t.getLeadSelectionPath());
  }
});
```

The listener is added using the `addTreeSelectionListener` method of `JTree`. Another way to do this is to register with the selection model itself:

```
TreeSelectionModel m = t.getSelectionModel();
m.addTreeSelectionModel(new TreeSelectionListener() {
```

In the event handler, the various paths are extracted and displayed, along with the flag indicating whether they were added or removed from the selection. To see some typical events, run the program and open the tree by expanding the `Apollo` node, the `11` node, and the `12` node; you should now have 16 rows on the screen.

First, select `Neil Armstrong`. Not surprisingly, this generates an event in which all the entries refer to this node:

```
First path: [null, Apollo, 11, Neil Armstrong]; added?
true
Lead path: [null, Apollo, 11, Neil Armstrong]
Old Lead path: null
Path: < [null, Apollo, 11, Neil Armstrong]; added? True
```

Next, click `Buzz Aldrin`, causing this entry to be selected and the previous one deselected. This time, the event contains entries for both paths:

```
First path: [null, Apollo, 11, Buzz Aldrin]; added? True
Lead path: [null, Apollo, 11, Buzz Aldrin]
Old Lead path: [null, Apollo, 11, Neil Armstrong]
Path: < [null, Apollo, 11, Buzz Aldrin]; added? True
Path: < [null, Apollo, 11, Neil Armstrong]; added? false
```

Here, the first path and the lead path are the one that has just been added, the old lead path is the lead path from the previous event, and the complete set of paths affected contains both the selected and the deselected path, together with booleans that indicate which of the two has just been selected. As long as you can be sure that only one item is being selected at a time (for example, if you use single-selection mode), each event will be self-contained and will tell you all you need to know about the state of the selection, as this one does. Things aren't always so simple, however. Now that you've got one path selected, hold down the `CTRL` key and click on `Pete Conrad`. This gives a selection of two individual items, but the event that it generates is not so obviously informative:

```
First path: [null, Apollo, 12, Pete Conrad]; added? True
Lead path: [null, Apollo, 12, Pete Conrad}
Old Lead path: [null, Apollo, 11, Buzz Aldrin}
Path: < [null, Apollo, 12, Pete Conrad]; added? True
```

You can see that `Pete Conrad` has just been added to the selection and that `Buzz Aldrin` must also be in the selection because that entry used to be the leading path, but note that the complete set of paths reported doesn't include `Buzz Aldrin`. This is because a `TreeSelectionEvent` only reports changes to the selection—anything that doesn't change is not mentioned. To emphasize this further, hold down `CTRL` and click `Alan Bean`. This gives three selected items and the following event:

```
First path: [null, Apollo, 12, Alan Bean]; added? True
```

```
Lead path: [null, Apollo, 12, Alan Bean]
Old Lead path: [null, Apollo, 12, Pete Conrad]
Path: < [null, Apollo, 12, Alan Bean]; added? true
```

Now, there's no mention of `Buzz Aldrin` at all and the set of paths still contains only the one entry. This is, of course, because only one path changed state.

Using the mouse with no modifiers or with the `CTRL` key pressed always generates events with only either one or two entries in the paths list. You can get events with more entries if you use the `SHIFT` key to select a range of items. To see an example, hold down the `SHIFT` key and click on `Apollo`. The `SHIFT` key selects all items between itself and the anchor of the existing selection. In this case, the selection will contain the items `Apollo`, `11`, `Neil Armstrong`, `Buzz Aldrin`, `Michael Collins`, `12`, `Pete Conrad` and `Alan Bean`, the last of which was the anchor selection before the selection changed. Recall that the anchor selection is the same as the lead selection provided that the `SHIFT` key has not been used.

Here is the event that is created for this selection:

```
First path: [null, Apollo]; added? true
Lead path: [null, Apollo]
Old Lead path: [null, Apollo, 12, Alan Bean]
Path: < [null, Apollo]; added? true
Path: < [null, Apollo, 11]; added? true
Path: < [null, Apollo, 11, Neil Armstrong]; added? true
Path: < [null, Apollo, 11, Michael Collins]; added? true
Path: < [null, Apollo, 12]; added? true
Tree's lead selection path is [null, Apollo]
```

As expected, the node that was clicked on, `Apollo`, is now the lead path and is listed as having been added, along with the nodes for `Neil Arm-strong` and `Michael Collins`.

So far, the lead path has always been the last path that was selected and this is usually the case, but there is an exception. Hold down the `SHIFT` key again and click on `13`, to create another block selection. You might expect that `13` would now be the lead path, but the event says otherwise:

```
First path: [null, Apollo, 12, Richard Gordon]; added? true
Lead path: [null, Apollo, 12, Alan Bean]
Old Lead path: [null, Apollo]
Path: < [null, Apollo, 12, Richard Gordon]; added? true
Path: < [null, Apollo, 13]; added? true
Path: < [null, Apollo]; added? false
Path: < [null, Apollo, 11]; added? false
Path: < [null, Apollo, 11, Neil Armstrong]; added? false
Path: < [null, Apollo, 11, Buzz Aldrin]; added? false
Path: < [null, Apollo, 11, Michael Collins]; added? false
Path: < [null, Apollo, 12]; added? false
Path: < [null, Apollo, 12, Pete Conrad]; added? false
Tree's lead selection path is [null, Apollo, 12, Alan Bean]
```

The lead path is, in fact, `Alan Bean`, the path at the top of the block selection. For block selections, this appears to be the rule because, as you saw in the previous example, if you click above the selection with `SHIFT` held down, the path that was clicked on, which will be at the top of the new block, is the lead path.

Finally, note that a `TreeSelectionEvent` is delivered only if the selection changes. To see this, click on `Richard Gordon`. This clears the old block selection and selects just one entry. Now click on `Richard Gordon` again. This time, you don't get an event—there is nothing to report because nothing has changed. To see why this is important, double-click on `Apollo`. This delivers just one event. Double-click again and you get no event at all. Many programs allow the user to pick an object from a list by double-clicking: This isn't going to be possible if you rely on `TreeSelectionEvent`s to tell you that this has happened. We'll see below how to handle double-selection as a means of selecting an entry in a tree.

If you don't need (or want) to keep track of exactly which paths have been affected by an event, you can just treat the event as notification that something has changed and query the tree model for the selected paths, as long as the model is derived from `DefaultTreeModel`. The methods that you can use, which are also implemented by `JTree` for convenience, were shown earlier in Table 10-4.

Converting Between Locations and Tree Paths or Rows

To make it possible for the user to choose an item from the tree by double-clicking, you have to register a

listener for mouse events, detect a double click, and convert the mouse location to a tree path. Fortunately, `JTree` has a convenience method that converts coordinates to a path in the tree, from which, as you know, you can get the node itself if you need it. Here's an example of the code that you would use to implement this:

```
// Here, 't' is a JTree. It must be declared as final
// so that we can use it in the event handler —
//     final JTree t;
t.addMouseListener(new MouseAdapter() {
  public void mouseClicked(MouseEvent evt) {
    if (evt.getClickCount() == 2) {
      // A double click — get the path
      TreePath path =
         t.getPathForLocation(evt.getX(), evt.getY());

      if (path != null) {
       Object comp = path.getLastPathComponent();
       if (t.getModel().isLeaf(comp)) {
         System.out.println("Selected path is " +
                         path);
       }
      }
    }
  }
});
```

Once the `TreePath` has been obtained, you can take any action that you need to in order to respond to the user's gesture. In this case, the code checks that the node being clicked on is a leaf node, so as not to confuse the use of double-clicking on a non-leaf node to open or close it with this use as a selection mechanism. Note that, before testing whether the chosen object is a leaf, a check is made that the `getPathForLocation` method returns a non-`null` `TreePath`. This is necessary, because the user could click inside the tree but outside of any area occupied by a tree node.

There are several methods that can be used to map between coordinates and parts of the tree and vice versa. These methods are summarized in Table 10-5.

Table 10-5 Mapping between coordinates and JTree

Method	Definition
`public TreePath getClosestPathForLocation(int x, int y)`	Gets the `TreePath`object for the node closest to the given position in the tree.
`public int getClosestRow-`	Gets the row number of the node closest
`ForLocation(int x, int y)`	to the given position in the tree.
`public Rectangle getPath-Bounds(TreePath path)`	Gets a `Rectangle`describing the area bounding the node corresponding to the given `TreePath`. The coordinates are relative to the top-left hand corner of the tree.
`public TreePath getPath-ForLocation(int x, int y)`	Gets the `TreePath`for the node that occupies the given coordinates

	relative to the top left of the tree. Returns `null` if there is no node at the given location.
`public TreePath getPath-ForRow(int row)`	Converts a row number to the `TreePath` for the node at that row.
`public Rectangle getRow-Bounds(int row)`	Gets a `Rectangle` describing the area bounding the node at the given row. The coordinates are relative to the top-left hand corner of the tree.
`public int getRowForLocation(int x, int y)`	Gets the row number for the node that occupies the given coordinates relative to the top left of the tree.

Traversing a Tree

Sometimes it is necessary to be able to traverse some or all of a tree, or to be able to search a subtree. The `TreeNode` interface allows you to find the parent of a given node and to get an enumeration of all of a node's children, which is all you need to implement your own searching mechanism. If all you can be sure of is that your tree is populated with `TreeNode`s (which is always a safe assumption), then you will have to be satisfied with the rather primitive `get-Parent` and `children` methods. On the other hand, if your tree is composed of `DefaultMutableTreeNode`s, you can make use of the following more powerful methods to traverse the tree or a subtree in various different orders:

```
public Enumeration pathFromAncestorEnumeration(
                      TreeNode ancestor)
public Enumeration preorderEnumeration()
public Enumeration postorderEnumeration()
public Enumeration breadthFirstEnumeration()
public Enumeration depthFirstEnumeration()
```

The `pathFromAncestorEnumeration` method is a little different from the other four, so we deal with it separately. This method involves two nodes—the one against which it is invoked (the target node) and the one passed as an argument, which must be an ancestor of the first node (if it is not, an `IllegalArgumentException` is thrown). Assuming that the ancestor is valid, this method walks the tree between it and the target node, adding to the enumeration each node that it encounters on the way. The first entry in the enumeration is the ancestor given as an argument and the last item is the target node itself. The other items in the enumeration are returned in the order in which they were encountered in the tree—in other words, each node is the parent of the one that follows it in the enumeration.

Of the other four enumerations, `depthFirstEnumeration` is the same as `postOrderEnumeration`. To see how the others order the nodes in the subtree that they cover, Listing 10-2 shows a short program that creates a small tree and applies `preorderEnumeration`, `postorderEnumeration`, and `breadthFirstEnumeration` to its root node. You can run this program for yourself by typing:

```
java JFCBook.Chapter10.TreeEnumerations
```

Listing 10-2 Various tree traversal enumerations

```
package JFCBook.Chapter10;
```

```
import javax.swing.*;
import javax.swing.tree.*;
import java.util.*;

public class TreeEnumerations {
  public static void main(String[] args) {
    JFrame f = new JFrame("Tree Enumerations");
    DefaultMutableTreeNode rootNode =
                    new DefaultMutableTreeNode("Root");
    for (int i = 0; i < 2; i++) {
    DefaultMutableTreeNode a =
                      new DefaultMutableTreeNode("" + i);
    for (int j = 0; j < 2; j++) { DefaultMutableTreeNode b =
                      new DefaultMutableTreeNode("" + i + "_" + j);
        for (int k = 0; k < 2; k++) {
          b.add(new DefaultMutableTreeNode(
                        ("" + i + "_" + j + "_" + k));
        }

        a.add(b);
      }
      rootNode.add(a);
    }

    JTree t = new JTree(rootNode);
    f.getContentPane().add(new JScrollPane(t));
    f.setSize(300, 300);
    f.setVisible(true);

    // Now show various enumerations
    printEnum("Preorder", rootNode.preorderEnumeration());
    printEnum("Postorder",
                     rootNode.postorderEnumeration());
    printEnum("Breadth First",
                      rootNode.breadthFirstEnumeration());
  }

  public static void printEnum(String title,
                               Enumeration e) {
    System.out.println("===============\n" + title);
    while (e.hasMoreElements()) {
      System.out.print(e.nextElement() + " ");
    }
    System.out.println("\n");
  }
}
```

The tree that this program generates is shown in Figure 10–9; the output follows. Here is what a preorder enumeration produces:

```
===============
Preorder
Root 0 0_0 0_0_0 0_0_1 0_1 0_1_0 0_1_1 1 1_0 1_0_0 1_0_1 1_1 1_1_0 1_1_1
```

As you can see, the preorder enumeration starts at the root and walks down the tree as far as it can go, adding the nodes that it traverses (0, 0_0, 0_0_0) to the enumeration. Having reached the bottom of the tree, it then walks across the children of the last node it traversed and adds them (0_0_1). Next, it moves back up to 0_0 and across to its sibling 0_1, which it adds to the enumeration and then descends and traverses *its* children (0_1_0, 0_1_1). This completes the traversal of the `0` subtree. Next, the subtree rooted at `1` is scanned in the same order.

Figure 10–9 A tree to demonstrate tree traversal enumerations

The postorder enumeration is very similar, except that it adds the parent nodes that it traverses as it crosses them on the way up, not on the way down. In other words, whereas the preorder traverse announced a subtree and then added its contents, postorder enumeration adds the contents and then adds the root of the subtree. If you think in terms of a file system, this is like processing all of the files in a directory and then applying the operation to the directory itself, which is the required order for removing a directory and its contents (since the directory must be empty to be removed).

```
===============
Postorder
0_0_0 0_0_1 0_0 0_1_0 0_1_1 0_1 0 1_0_0 1_0_1 1_0 1_1_0 1_1_1 1_1 1 Root
```

Breadth-first enumeration is much simpler to visualize—it just walks across all the nodes at each level and adds them to the enumeration, then goes down a level and lists all of the nodes at that level, and so on. In terms of the tree's display, if you open all of the subtrees so that every node in the tree is visible, you'll see that this enumeration walks vertically down the screen and, when it reaches the bottom, goes back up and across to the right one level, then back down the screen again, and so on. The effect is that you see everything at depth 0, then everything at depth 1, followed by depth 2 and so on.

```
===============
Breadth First
Root 0 1 0_0 0_1 1_0 1_1 0_0_0 0_0_1 0_1_0 0_1_1 1_0_0 1_0_1 1_1_0 1_1_1
```

Changing Tree Content

Some trees will be static once you've created them, but many will not. If you need to change the content of the tree's data model, there are two ways you can do it—at the model level and at the node level. You need to handle these two possibilities slightly differently, so they are described separately in this section.

Making Changes via DefaultTreeModel

The generic `TreeModel` interface doesn't offer any way to change the content of the model—it assumes that any changes will be made at the node level. The problem with just changing the nodes is that the screen display of the tree won't be updated. It isn't sufficient just to make changes to the data—the tree's user interface class must also be told about these changes.

If you want to handle changes at the node level, you'll see what you need to do in the next section. If you don't want to go to that much trouble, however, there are two convenience methods in the `DefaultTreeModel` that do everything for you:

```
public void insertNodeInto(MutableTreeNode child,
          MutableTreeNode parent, int index)
public void removeNodeFromParent(MutableTreeNode parent)
```

These methods make the changes that their names imply by manipulating the node data in the model, then they arrange for the tree's on-screen representation to be updated by invoking the `nodesWereInserted` and `nodesWereRemoved` methods of `DefaultTreeModel` that will be covered shortly. If you are making a small number of changes to the model, `insert-NodeInto` and `removeNodeFromParent` are the simplest way to do it. However, because these methods generate one event for each node you add or remove, if you are adding and changing more than a few nodes, it can be more efficient to side-step them and manipulate the nodes directly because, by doing so, you can generate fewer events.

Making Changes at the Node Level

`DefaultMutableTreeNode` has six methods that allow you to make changes to the data model by directly manipulating the nodes themselves:

```
public void add(MutableTreeNode child)
public void insert(MutableTreeNode child, int index)
public void remove(int index)
public void remove(MutableTreeNode child)
public void removeAllChildren()
public void removeFromParent()
```

With the exception of the last, all these are invoked against the parent to be affected by the change; `removeFromParent` is invoked on the child node to be removed from its parent. The effect of these methods is obvious from their names. All that they do, however, is to update the tree's data model and keep it in a consistent state. They do not inform the tree itself that it may need to redisplay its contents. After making any changes to the node structure, you need to invoke one of the following `DefaultTreeModel` methods to ensure that the tree's appearance matches its internal data:

```
public void reload()
public void reload(TreeNode node)
public void nodesWereInserted(TreeNode node,
                      int[] childIndices)
public void nodesWereRemoved(TreeNode node,
```

```
    int[] childIndices, Object[] removedChildren)
public void nodeStructureChanged(TreeNode node)
```

The first of these methods is the most radical solution: It tells the `JTree` to discard everything that it has cached about the structure of the tree and to rebuild its view from scratch. This could be a very slow process and is not recommended except when you really have replaced the entire data model. The second form of the `reload` method gives a node as a limiting point. When this method is used, the tree assumes that there may have been radical changes but that they were confined to the subtree rooted at the node given as the argument. In response, the tree rebuilds its view from that point in the hierarchy downward. This method is ideal if you make major changes under a node, such as populating it from scratch or removing all of its children. You might, for example, use this after calling `removeAllChildren` and pass the node whose children were removed as its argument:

```
// Delete everything below "node" and tell the
//tree about it
node.removeAllChildren();
model.reload(node);   // "model" is the DefaultTreeModel
```

The method `nodeStructureChanged` is identical to `reload`.

The other two methods are used for finer control over the mechanism. You can use these to minimize the impact of any changes by informing the tree exactly which nodes were added or removed. Both of these methods require you to accumulate a list, in ascending numerical order, of the indices of all of the nodes that were added or removed and, in the latter case, a list of all of the removed nodes themselves. This is obviously more complex than just using `nodeStructureChanged`, but it can be more efficient. These methods have the limitation that they can only inform the tree about changes in one parent node at a time. If you make changes to more than one node, for example, to several levels in the hierarchy, you need to invoke these methods once for each parent node affected, which can increase the complexity of the task, but this may be justified by the performance gains.

Changing the Attributes of a Node

There is a third type of change to the tree model that hasn't been covered so far—changes to the contents of the nodes themselves. Suppose that you make a change to a node that doesn't affect its relationship to its neighbors but should result in its appearance changing. This might happen if you change the value of the node's user object, as might be the case if your tree represents a file system and you allow the user to rename a file or directory. When this happens, you can use one of the following `DefaultTreeModel` methods:

```
public void nodeChanged(TreeNode changedNode);
public void nodesChanged(TreeNode parentNode,
                    int[] childIndices)
```

Notice the important difference between these two methods: When you change one node, you pass a reference to that node, but if you change several, then you must pass a reference to their common parent and a set of indices that identify the affected children. If you make changes that affect nodes in several parts of the tree, then you must invoke `nodesChanged` once for each node that has children that have been changed.

Tree Model Events

All of the methods in the previous sections operate by generating `TreeMod-elEvent`s that can be caught by registering a `TreeModelListener` on the tree's `DefaultTreeModel` object. Usually, Swing and AWT events are not highly specific about exactly what has changed—the listener is expected to query the source of the event to find out what happened. This is acceptable for simple controls like lists, combo boxes, text fields, and buttons where only one thing can have happened. But in the case of a tree, many modifications of different types can be made. If a single event that said "something has changed" were posted, it would be necessary to scan the whole tree to try and detect what happened. This would, of course, be time-consuming and would require the listener to remember a large amount of state in order to be able to perform a comparison at all.

Because of this, the `TreeModelListener` must implement four methods, each of which receives a `TreeModelEvent`:

```
public void treeNodesChanged(TreeModelEvent)
```

```
public void treeStructureChanged(TreeModelEvent)
public void treeNodesInserted(TreeModelEvent)
public void treeNodesRemoved(TreeModelEvent)
```

These listener methods are entered as a result of following `Default-TreeModel` methods being called:

treeNodesChanged	nodeChangedor nodesChangedwas called
treeStructureChanged	reload(either variant) or nodeStructureChanged was called
treeNodesInserted	nodesWereInserted was called
treeNodesRemoved	nodesWereRemovedwas called

Splitting at this level is not sufficient, however, because it doesn't tell the listener what was added, removed, or changed. This information is contained in the `TreeModelEvent`, which has five methods that can be used to extract details of the change to the model:

```
public Object getSource()
public TreePath getTreePath()
public Object[] getPath()
public Object[] getChildren()
public int[] getChildIndices()
```

Which of these methods returns meaningful values and what they mean depends on the type of event. All of the events return a valid value to `get-Source`, which is a reference to the affected tree model (i.e., the `Default-TreeModel` instance for the affected `JTree`). The `getTreePath` and `getPath` method both return the node in the tree that is affected by the change, either in terms of its `TreePath` or as the array of `Object`s that makes up the `TreePath`. In the case of an event delivered to `treeStructureChanged`, only `get-Path` and `getTreePath` return any useful information. These methods return the node and below which the change occurred. This `TreePath` corresponds to the node that was originally passed to `reload` or `nodeStructure-Changed`. The `getChildren` and `getChildIndices` methods both return `null`.

For `treeNodesChanged`, the values returned by `getPath` and `getTree-Path` again correspond to the node that is the parent of the node or nodes that were affected. The `getChildIndices` and `getChildren` methods return arrays that contain one entry for each affected child of that node. The indices in the array returned by `getChildIndices` are in ascending numerical order and the array returned by `getChildren` is sorted so that corresponding entries in these two arrays match—that is, the index in entry `n` of the index array corresponds to the child in entry `n` of the child array.

Similarly, `getPath` and `getTreePath` return the parent of the affected nodes in both `treeNodesRemoved` and `treeNodesInserted` and the `get-ChildIndices` and `getChildren` methods retrieve sorted and matching arrays of the affected children. Note, however, that in the case of `treeNodes-Removed`, these children have already been removed from the parent node and the indices indicate where they used to reside in the parent's list of children, whereas for `treeNodesInserted`, these indices describe where the children have been inserted—that is, they describe the final state of the model.

Most applications won't need to handle these events because the `JTree` look-and-feel class registers itself as a listener of the tree's model and updates the tree's appearance based on the events that it receives. You can, however, register a listener of your own if you want the tree control to cause your application to react in some special way to changes in its structure. You'll see examples of this in the rest of this chapter.

A File System Tree Control

The most obvious thing to use a tree control for is to show the content of a machine's file system, in the same way as the Windows Explorer does in the left of its two display panes. Since the file system and `JTree` are both inherently hierarchical, you might expect it to be a simple matter to create a tree control that maps a file system but, as you'll see in this section, there are a few interesting issues to tackle along the way. As the description of the implementation of this component progresses, you'll see some of the real-world problems that come up when creating new components, including the need to sometimes work around problems in the classes that the new control is based on.

Before looking at how to implement a file system control, let's look at what the finished product looks like. Figure 10-1 shows the control mounted in a frame and running on a Windows platform. If you want to try this out for yourself, use a command like the following:

```
java JFCBook.Chapter10.FileTreeTest C:\
```

You can replace the string "C:\" with the name of any directory on your system—this is the directory that becomes the root of the file tree.

When you run this example, you should see the directory whose name was supplied on the command line at the top of the display, followed by all of its subdirectories. Like Windows Explorer, this control only shows directories, not files. Windows Explorer shows files in its right-hand pane, which is almost identical to the Swing `JFileChooser`. Indeed, once you have this control, you can use it, together with the `JFileChooser` and the `JSplit-Pane` component that you saw in the last chapter to create an application that looks very much like Windows Explorer itself except, of course, that it also runs on many versions of UNIX (and other platforms).

If you move the mouse over any directory with an expansion icon to the left of it and click on the icon or double-click on the directory icon or name, it opens to reveal its subdirectories. If those subdirectories have other subdirectories, they will also have expansion icons next to them. A directory that doesn't have any subdirectories (i.e., a directory that is empty or contains only files) cannot be expanded. Again, to see the contents, you would need to connect this control to a `JFileChooser` and have it display the directory's contents.

Core Note

The Swing `JTree` component doesn't work ideally with respect to displaying the icon for a node that has no children. If you're using a look-and-feel with this problem, you'll find that all of the subdirectories seem to have contents but, when you click on one that actually only has files in it, the icon sign disappears. All of the standard look-and-feels currently behave this way.

Implementation Plan for the File System Control

In order to create a tree that shows a file system, a representation of the file system's directory structure must be loaded into a `DefaultTreeModel`. On the face of it, this is a relatively simple proposition: All you need is a recursive method that is given a directory, reads its contents and, each time it comes across a subdirectory, adds a node to the tree model, then invokes itself to process the content of that subdirectory. Using the methods of the `java.io.File` class, it is easy to get a list of the objects in a directory and to determine which of these objects are themselves directories. The problem with this simplistic approach is that it would scan the entire file system when the control is created. This has two obvious drawbacks—it may take a long time and it will probably take up a large amount of memory to hold all of the nodes. Most of this time and memory is probably wasted, because the odds are that the user will find what he/she is looking for fairly quickly and not look at most of what has been generated.

The ideal approach would be to start by doing only as much work as necessary to show the user the content of the directory that the control starts in. Then, whenever a directory is clicked and opened, the nodes for that directory should be created and displayed, and so on. If this can be achieved, it should be possible to create the control with only the relatively short delay required to read the top-level directory, at the cost of a small delay each time a new directory is opened. This approach is, however, too simplistic.

To see why this won't work, suppose you have a small file system with a directory structure that looks like this:

```
C:\java
C:\java\bin
C:\java\demo
C:\java\demo\Animator
   C:\java\demo\ArcTest
   C:\java\demo\awt-1.1
   C:\java\demo\BarChart
C:\java\docs
   C:\java\docs\api
   C:\java\docs\guide
   C:\java\docs\images
   C:\java\docs\relnotes
   C:\java\docs\tooldocs
C:\java\include
   C:\java\include\win32
C:\java\lib
C:\java\src
   C:\java\src\java
     C:\java\src\java\applet
     C:\java\src\java\awt
     C:\java\src\java\bean
     C:\java\src\java\io
   C:\java\src\sun
   C:\java\src\sunw
```

To create a tree that allows the user to navigate the directory `C:\java` following the algorithm outlined earlier, the first step would be to build a node for `C:\java` itself and then child nodes would be added for the subdirectories `C:\java\bin`, `C:\java\demo`, `C:\java\docs`, `C:\java\include`, `C:\java\lib` and `C:\java\src`. If the process stopped here, the tree would indeed show the top-level directory and all of these subdirectories. However, you can see that the `demo`, `docs`, `include`, and `src` directories themselves have subdirectories, so you would expect the tree to show them with an expansion icon. But, this icon is only shown if the node that it relates to has child nodes attached to it. So far, no tree nodes have been added for these subdirectories, because the idea is to minimize the work that is needed to get the control created. To get the right effect, this approach will have to be modified in some way.

The only way to make the expansion icons appear is to add a child node to each directory that has subdirectories. To do this, it is necessary to scan not only the directory that the control starts in, but also the directories that are found in that directory. With this modification, nodes would be added for the subdirectories `Animator`, `ArcTest`, `awt-1.1`, `BarChart`, `api`, `guide`, and so on. It isn't necessary to go any lower at this stage, because directories further down can't influence the appearance of the control when the user is only looking at the contents of the `C:\java` directory.

Now suppose the user clicked on the icon for `C:\java\src`. This would open up and reveal the nodes representing the `java`, `sun` and `sunw` subdirectories, but now there is the same problem again: The `C:\java\src\java` subdirectory and (although this hasn't been shown in detail in the file system map shown above) all the other directories have subdirectories of their own, but the expansion icons for their nodes don't appear because those directories haven't been scanned to build their child nodes. Fortunately, as you know, it is possible to get a `TreeExpansionEvent` when the user opens a node. This event can be used to find out which directory the user wants to look at and then all of its subdirectories can be populated, as was done with the initial directory. In fact, as you'll see, the same code can be used to handle both cases.

The approach just outlined will work, but there is a small improvement that can be made to minimize the impact of having to scan for subdirectories. The only reason to look for directories that are not going to be visible straight away is to make the expansion icon appear for a directory that will be visible. All you have to do to make the expansion icon appear is to place *one* child under the node for a directory that has subdirectories. In other words, when scanning for hidden subdirectories, it is possible to stop as soon as we have found one. This can save a lot of time if the file system has big directories, because it avoids the overhead of reading through lots of files to find subdirectories that may not exist, and also minimizes the effect of looking in directories that the user doesn't end up opening anyway. The flip side of this is, of course, that if the user does open a directory that has had a single entry placed in it, it is necessary to get rid of it before fully populating the directory, or the first entry wold appear in the directory twice.

File System Tree Control Implementation

Having seen the basic idea, let's now examime the implementation, the code for which appears in Listing 10-3.

Listing 10-3 A custom control to display a file system in a tree

```
package JFCBook.Chapter10;

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.io.*;
import java.util.*;

public class FileTree extends JTree {
  public FileTree(String path)
      throws FileNotFoundException, SecurityException {
  super((TreeModel)null);// Create the JTree itself

  // Use horizontal and vertical lines
  putClientProperty("JTree.lineStyle", "Angled");

  // Create the first node
  FileTreeNode rootNode = new FileTreeNode(null, path);

  // Populate the root node with its subdirectories
  boolean addedNodes = rootNode.populateDirectories(true);
  setModel(new DefaultTreeModel(rootNode));

  // Listen for Tree Selection Events
  addTreeExpansionListener(new TreeExpansionHandler());
}

// Returns the full pathname for a path, or null
// if not a known path
public String getPathName(TreePath path) {
  Object o = path.getLastPathComponent();
  if (o instanceof FileTreeNode) {
     return ((FileTreeNode)o).file.getAbsolutePath();
  } return null;
}

// Returns the File for a path, or null if not a known path
public File getFile(TreePath path) {
  Object o = path.getLastPathComponent();
  if (o instanceof FileTreeNode) {
    return ((FileTreeNode)o).file;
  }
  return null;
}

// Inner class that represents a node in this
// file system tree
protected static class FileTreeNode
              extends DefaultMutableTreeNode {
  public FileTreeNode(File parent, String name)
      throws SecurityException, FileNotFoundException {
  this.name = name;

  // See if this node exists and whether it
  // is a directory
  file = new File(parent, name);
  if (!file.exists()) {
    throw new FileNotFoundException("File " + name + " does not exist");
  }

  isDir = file.isDirectory();

  // Hold the File as the user object.
  setUserObject(file);

}

// Override isLeaf to check whether this is a directory
public boolean isLeaf() {
  return !isDir;
}

// Override getAllowsChildren to check whether
// this is a directory
public boolean getAllowsChildren() {
  return isDir;
}
```

```
// For display purposes, we return our own name public String toString() { return name; }
// If we are a directory, scan our contents and populate
// with children. In addition, populate those children
// if the "descend" flag is true. We only descend once,
// to avoid recursing the whole subtree.
// Returns true if some nodes were added
boolean populateDirectories(boolean descend) {
  boolean addedNodes = false;
// Do this only once
if (populated == false) {
  if (interim == true) {
    // We have had a quick look here before:
    // remove the dummy node that we added last time
    removeAllChildren();
    interim = false;
  }

  String[] names = file.list();// Get list of contents

  // Process the directories
  for (int i = 0; i < names.length; i++) {
    String name = names[i];
    File d = new File(file, name);
    try {
      if (d.isDirectory()) {
        FileTreeNode node =
                new FileTreeNode(file, name);
        this.add(node);
        if (descend) {
          node.populateDirectories(false);
        }
        addedNodes = true;
        if (descend == false) {
          // Only add one node if not descending
          break; }
      }
    } catch (Throwable t) {
      // Ignore phantoms or access problems
    }
  }
// If we were scanning to get all subdirectories,
// or if we found no subdirectories, there is no
// reason to look at this directory again, so
// set populated to true. Otherwise, we set interim
// so that we look again in the future if we need to
if (descend == true || addedNodes == false) {
  populated = true;
} else {
// Just set interim state
        interim = true;
      }
    }
    return addedNodes;
  }

  protected File file;// File object for this node
  protected String name;// Name of this node
  protected boolean populated;
                    // true if we have been populated
  protected boolean interim;
                    // true if we are in interim state
  protected boolean isDir;// true if this is a directory
}

// Inner class that handles Tree Expansion Events
protected class TreeExpansionHandler
                  implements TreeExpansionListener {
  public void treeExpanded(TreeExpansionEvent evt) {
    TreePath path = evt.getPath();// The expanded path JTree tree =
    (JTree)evt.getSource();// The tree

    // Get the last component of the path and
    // arrange to have it fully populated.
    FileTreeNode node =
               (FileTreeNode)path.getLastPathComponent();
    if (node.populateDirectories(true)) {
      ((DefaultTreeModel)tree.getModel()).
                        nodeStructureChanged(node);
}
```

```
    }

    public void treeCollapsed(TreeExpansionEvent evt) {
      // Nothing to do
    }
  }
}
```

Naturally enough, the control, called `FileTree`, is derived from `JTree`. It is a self-contained component that is given a directory name as its starting point—that's all you need to do to get a hierarchical display of the objects below that directory. The component can be placed in a `JScrollPane` and used just like any other tree.

Designing the Data Model

The constructor first invokes the `JTree` constructor, creating a tree with no data model, then starts building the contents of the control by creating a `DefaultTreeModel` that initially contains just a node for the starting directory. The obvious question that arises is whether to use a `DefaultMutable-TreeNode` to represent the directories, or to create a node class of our own.

This turns out to be a pretty simple decision. Suppose we chose to use a `DefaultMutableTreeNode`. Along with each node, we need to store something that connects it to the object in the file system that it represents. The most natural way to do this is to use the file or directory's `File` object and store it as the user object of the `DefaultMutableTreeNode`. However, simply populating the model with `DefaultMutableTreeNode`s set up in this way would not work, because the tree uses the `toString` method of a node's user object to obtain the text to be displayed for that node. In this case, we need the tree to display the last component of the file name, which is not what the `toString` method of `File` returns. Instead, we create a private node class to represent each object in the part of the file system that the tree will display. Because this class is tied to the design of the tree, it is implemented as an inner class of the tree, called

`FileTreeNode` and is derived, naturally, from `DefaultMutableTreeNode`. This class will store the `File` object and the node's name relative to its parent, both of which will be passed as arguments to the constructor. To achieve the correct display, the `toString` method will return the latter of these two values. It turns out that we wouldn't actually need to store the `File` as the node's user object, but we choose to do so anyway so that an application that listens to selection events from the `FileTree` component can get easy access to the corresponding `File`.

Building the Initial Data Model

Construction of the data model begins with the node for the initial directory, to which other nodes must be added for each of its immediate subdirectories. This task is delegated to a method of `FileTreeNode` called `populateDi-rectories`, which you'll see in a moment. When this method returns, a `DefaultTreeModel` initialized with the root node is created and the `JTree setModel` method is used to attach it to the tree. The model as created by this method is sufficient for the initial display of the tree and will remain suf-ficient until the user (or the application) expands one of the nodes. To handle this (likely) eventuality, the last thing the `FileTree` constructor has to do is register a listener for `TreeExpansionEvent`s. The code that will handle these events is encapsulated in a separate inner class called `TreeExpan-sionHandler`, to avoid cluttering `FileTree` with methods that aren't part of its public interface. The constructor for `FileTree` creates an instance of this class and registers it to receive the events. At this point, the control is fully initialized and ready to be displayed.

Apart from its constructor, `FileTree` only has two methods of its own. The `getPathName` method is a convenience method that accepts a `TreePath` as its argument and returns the path name of the file that it represents in the tree. Similarly, the `getFile` method accepts a `TreePath` argument and returns the `File` object for the file that it represents.

Once the tree model has been created, it is managed entirely by the `JTree` class. All that remains for us to do is implement the inner classes `FileTreeN-ode` and `TreeExpansionHandler`.

Implementing the FileTreeNode Class

The `FileTreeNode` class is derived from `DefaultMutableTreeNode` so that it can be added to a tree. Its constructor has two arguments that give the `File` object for the directory that it resides in and name of the object that it represents relative to that directory. These values are combined to create a `File` object for the target file or directory, which is stored in both the instance variable `file` and as the user object of the node. For convenience, the file name passed to the constructor is also stored in the instance variable `name`, from which it can be quickly retrieved by the `toString` method. The case of the root node (the one that represents the initial directory) is a special one, because it does not have a parent `File` object. In this case, the `File` argument can be supplied as `null`.

All that remains for the constructor to do is to determine whether the node's associated object in the file system is a directory or not. This is easily done be calling the `isDirectory` method of the newly constructed `File` object. The result of this call is stored in an instance variable called `isDir`, of type `boolean`. Unfortunately, this method returns `false` if the object is not a directory or if it doesn't exist. Most of the time, objects will be constructed using names that have been discovered by reading the file system, which (subject to possible race conditions) should exist, but the node at the top of the tree will be built based on information supplied by the user, so it might not exist. To be safe, the existence of the object is verified by using the `exists` method and a `FileNotFoundException` is thrown if necessary. On some systems (e.g., UNIX and Windows NT), it may turn out that the user of the control doesn't have access to the directory in which the object resides, or to the object itself. On these systems, the constructor will throw a `Securi-tyException`. If either of these exceptions occurs while the tree is being built, they are just caught and ignored, with the result that the tree will be properly constructed but won't contain those objects that it couldn't get access to. The exception to this is the root node, which is built in the `File-Tree` constructor: Any problem creating this is passed as an exception to the code creating the tree.

Core Note

A nice enhancement to this control might be to catch a `SecurityException` and represent the object that could not be accessed in some meaningful way to the user—perhaps with an icon representing a `NO ENTRY` sign. You shouldn't find it too difficult, as an exercise, to add this functionality.

FileTreeNode Icon and Text Representation

Before looking in detail at how to handle the population of subdirectories, there is a small point that needs to be dealt with. `JTree` has a small selection of icons that it can display for any given node—a sheet of paper representing a leaf, a closed folder, and an open folder (in the Metal look-and-feel), both of which are used for branch nodes. As you know, there are two different ways to determine whether a node considers itself to be a leaf, only one of which is usually appropriate for a particular type of node: Either its `isLeaf` method or its `getAllowsChildren` method should be called. In the case of a file system, directories are never leaves, even if they are empty, so it would be normal to implement the `getAllowsChildren` method and have it return `false` if the node represents a directory and `true` otherwise. For flexibility, the `FileTreeNode` class also implements the `isLeaf` method, which returns the inverse of whatever is returned by `getAllowsChildren`. Since both of these return correct answers for any node, there is no need to tell the `DefaultTreeModel` which method to use, so the default setting is used when the data model is created (which means that `isLeaf` will be called).

The text that is rendered alongside the icon is obtained by invoking the `toString` method of the object that represents the node—that is, the `toString` method of `FileTreeNode`—so, as noted earlier, this method is overridden to return the name of the component.

Populating Directory Nodes

The last important piece of the `FileTreeNode` class is the `populateDi-rectories` method, which contains the code to create nodes to represent subdirectories. As noted earlier, sometimes it is necessary to create nodes for all of a directory's subdirectories and sometimes only one node is needed so that the directory's icon shows that it can be expanded. In fact, these actions will always need to be carried out in pairs, one immediately after the other. To see why this is, consider the file system example that is shown above and imagine what needs to happen when the root node for the `C:\java` directory is created. The first thing to do is to get a list of the directories in the directory `C:\java`. Since all of these are going to be shown straight away, it is necessary to create nodes for each of them immediately and attach them to the root node. If any

of these directories has subdirectories, a single node must be attached to them, to show that the directory is expandable. So, for each directory that is created in `populateDirectories`, there will need a second scan that creates no more than one node under it. Since both of these operations involve scanning a directory and creating a node (or nodes) to attach to the directory's node, the same code can be used for each case.

For this reason, the `populateDirectories` method takes a boolean argument called `descend`. When it is called to populate a directory whose content is going to be displayed because its node is being expanded, this parameter is set to `true` and the entire directory will be scanned and populated. On the other hand, when it is called to scan a newly-discovered subdirectory which will be displayed as a collapsed node, `descend` will be set to `false`. If `descend` is `false`, the loop that processes each directory will terminate when it finds a subdirectory. Also, if `descend` is `false`, discovering a subdirectory doesn't cause `populateDirectories` to descend into it—which is the reason for the name. If this were not done, `populateDirecto-ries` would always recursively descend down to the end of one path on the file system, instead of scanning only two levels.

Let's see how this works in practice. Refer to the implementation of `pop-ulateDirectories` in Listing 10-3 as you read this. When this method is first called, `descend` is set to `true`. Ignore, for a moment, the `populated` and `interim` instance variables, which are both initially `false`—you'll see how these are used in a moment. The first thing to do is create a `File` object for the node and get a list of its contents using the `list` method.

Core Note

Notice that this code is very carefully arranged so that, if it gets a `SecurityException`, it returns having done no damage to the tree structure. This means that the tree will be incomplete, but will reflect those parts of the file system that the user has access to.

In the case of the example file system, this will return the following list of subdirectories:

```
bin, demo, docs, include, lib, src
```

It will also return the names of any files in the `java` directory; on my machine, there are several of these, but I've left them out of this list for clarity. A `FileTreeNode` for each of these directories must be added to the node that represents the directory being scanned. This job is carried out by the `for` loop, which makes one pass for each name in the list. Each time round the loop, it creates a `File` object for one of the above names (and the files that I haven't shown) and checks whether it represents a directory. If it doesn't, it is ignored and the next pass of the loop is started. In the case of a directory, it creates a `FileTreeNode`, passing the `File` object of the parent (which is the `File` object of the current node) and the component name from the list (i.e., `bin` or `demo`, etc.), then adds it underneath the node for the directory that is being scanned. Next comes the interesting part. If `descend` is true, which is the case when a directory node is being expanded, `populateDirectories` is called again, but this time there are two important differences:

- The `descend` argument changes to `false`.
- The node on which it operates is the new node (i.e., that for `bin`, `demo`etc.), not the top-level node.

This second call does the same things again for the subdirectory that it is dropping into, but it stops sooner—as soon as any subdirectory is found. This call is made for each of the directories in the original list (i.e., `bin`, `demo`, `docs`, `include`, `lib` and `src`). In the case of `bin`, there are no subdirectories, so it will complete the `for` loop without finding anything and just return (what happens to the `populated` and `interim` fields will be discussed below). However, the second pass of the loop calls `populateDirectories` for the `demo` node, which does have subdirectories. In this case, the `list` method returns the following set of names (and possibly some files that will be ignored because they have no bearing on how the data model is constructed):

```
Animator, ArcTest, awt-1.1, BarChart
```

Now the `for` loop is entered and the code starts handling the subdirectory `Animator`. This turns out to be a directory, so a `FileTreeNode` is built for it and attached under the node for `C:\java\demo`. However, this time, `descend` is `false`, so `populateDirectories` is not called again to descend into this new directory. Instead, the `for` loop ends, having added exactly one child to `C:\java\demo`. Control now returns to the `for` loop of the first invocation of `populateDirectories`, where the rest of the original list of names is processed in the same way but, because here `descend` is set to `true`, the loop runs to completion.

When this loop has been completed, the state of the `C:\java` node needs to be changed to reflect that fact that all of its subdirectories have been located and nodes have been built for them. To indicate this, the

`populated` field is set to `true`. When `populateDirectories` is called for a node with `populated` set to `true`, it knows that there is nothing more to do and just returns at once. As you'll see, this can happen during tree expansion. On the other hand, the nodes for the subdirectories `demo`, `docs`, `include`, and `src` have all been processed by `populateDirectories` and each has had one dummy node added. These directories are not completely processed and it will be necessary to scan them again later if the user expands them in the tree, so it would be incorrect to set `populated` to `true`. But it is also necessary to remember that a child has been attached to these nodes so that it can be discarded later before building a fully populated subtree. To distinguish this case, the instance variable `interim` is set to `true`, because this node is in an interim state. Note, however, that there is a special case here. If a scan of a subdirectory finds that it has no subdirectories at all, it is marked as populated, because there is nothing to be gained from scanning it again later.

At the end of this process, the following node structure will be as follows:

```
C:\java (populated)
   C:\java\bin (populated)
   C:\java\demo (interim)
      C:\java\demo\Animator
   C:\java\docs (interim)
      C:\java\docs\api
   C:\java\include (interim)
      C:\java\include\win32
   C:\java\lib (populated)
   C:\java\src (interim)
      C:\java\src\java
```

Note that the top-level directory is marked as populated, as are the subdirectories that don't themselves have any further directories (`bin` and `lib`). The others are marked as interim, because only one node has been created underneath them and more would need to be added before the user could be allowed to expand, say, `C:\java\docs`. Notice also that `C:\java\include` is marked as interim despite the fact that it only has one subdirectory and a node has been created for it. This is because `populateDirectories` stops after finding one directory, so it wasn't able to see that there weren't any more directories to find. The point of this optimization is the assumption that it is better to stop after finding one directory, on the grounds that if there is one subdirectory there is probably another and, if there isn't, there may still be a large number of files to traverse before getting to the end. Sometimes this guess will be right, other times it won't. In this case, a little more work than is strictly necessary will be done if the user opens this directory, because the node for `win32` will be discarded and then put back again.

Handling Expansion of Directories

In terms of the tree as seen by the user, the node structure is now correct and the tree will display the `demo`, `docs`, `include`, and `src` directories with an expansion icon and the others without one. Nothing will change until the user clicks on one of these icons. Suppose the user clicks on the node for `C:\java\demo`. Unless something is done, in response to this click the `JTree` will expand the node and show just the entry for `C:\java\demo\Animator`. This is, of course, wrong because there are three missing subdirectories. Here is where the `TreeExpansionListener` comes into play. When the user clicks on the node, the `treeExpanded` method of the `TreeExpansionHan-dler` inner class will be entered. The `TreeExpansionEvent` that it receives tells it the source of the event (i.e., the `FileTree` instance) and the `TreePath` for the node that was expanded—for instance a `TreePath` for `C:\java\demo`. To give the tree the correct appearance, three new nodes must be added under the node for `C:\java\demo`, for which it is necessary to find the `FileTreeNode` for that path.

As you saw earlier in this chapter, given a `TreePath`, you can find the `TreeNode` for that path by extracting the path object for the last component in the `TreePath`. To get the `FileTreeNode` for `C:\java\demo`, then, the `getLastComponent` method is invoked on the `TreePath` from the event. This returns an `Object` that is actually the `FileTreeNode` for the path. Armed with this, the node for `C:\java\demo` can be populated by invoking `populateDirectories` against it, passing the argument `true` to have it process two levels of directory, so that the new directories that get added will get the right expansion icons.

This time, `populateDirectories` will start at `C:\java\demo`, which is marked as interim, so it starts by discarding all of its children, which causes the existing node for `C:\java\demo\Animator` to be removed. Then, it adds nodes for all four subdirectories of `C:\java\demo` and marks it as populated. Along the way, it will have looked to see if there were directories in any of these four subdirectories and marked the new nodes for the `Animator`, `ArcTest`, `awt-1.1`, and `BarChart` directories as interim if they were or populated if there were not.

Finally, adding new nodes below an existing node is not sufficient to make the tree show them. To have the tree update its view, the model's `node-StructureChanged` method must be called, passing it the node for `C:\java\demo`, since this is the parent node beneath which all of the changes were made. The event handler has a reference to this node, but how is it possible to get hold of a reference to the model? There are two ways to do this. First, the `TreeExpansionHandler` is a nonstatic inner class of `FileTree`, which is a `JTree`, so the model reference can be obtained using the following expression:

```
FileTree.this.getModel()
```

This works because `FileTree.this` refers, by definition, to the containing `FileTree`. A simpler way, though, is to use the fact that a reference to the `JTree` is provided as the source of the `TreeExpansionEvent`, so you can instead just do this:

```
JTree tree = (JTree)evt.getSource();
((DefaultTreeModel)tree.getModel()).
                nodeStructureChanged(node);
```

This is the approach taken in the implementation.

Using the File System Tree Control

The file system tree control is intended to be a black box. To use it, you decide on the starting point in the file system, then create an appropriate `FileTree` object.Usually, you will want to know when the user has selected a file in the tree. Since `FileTree` is derived from `JTree`, it supports the same events, so you can receive notification of user actions by registering a listener for the `TreeSelectionEvent`, which, as you know, is generated when any change is made to the selection. What you really want to know when you get one of these events is the file name or the `File` that corresponds to the node that has been selected, but the event itself only has `TreePath` objects associated with it. As you've already seen, you can get the complete set of objects that make up a `TreePath` by calling its `getPath` method. Once you've got these objects, you can invoke `toString` on each of them, then join them with the system file name separator character to get the full path name of the file. However, this would be a very slow process, so the method `getPath-Name` was added to `FileTree`. This method takes a `TreePath` from an event and returns the full path name that it represents, or `null` if it isn't a path in the tree. The implementation of this method, which is repeated from Listing 10-3, is very simple:

```
// Returns the full pathname for a path, or null
// if not a known path
public String getPathName(TreePath path) {
  Object o = path.getLastPathComponent();
  if (o instanceof FileTreeNode) {
    return ((FileTreeNode)o).file.getAbsolutePath();
  }
  return null;
}
```

First, the object the represents the last component of the `TreePath` is obtained using `getLastPathComponent`. If this `TreePath` was really obtained from an event generated by `FileTree`, the user object will be the `FileTreeNode` for the selected file. Before casting it, however, the code ensures that this is the case to protect against the application passing erroneous paths to `getPathName`. If the path really does correspond to a `File-TreeNode`, the full path name of the selected file is obtained from the `File` object which was stored in the `file` field when the node was constructed, and returned to the caller. If this is not a `FileTreeNode`, `null` is returned. For the case in which the application needs the `File` object for the selected node, there is a similar convenience method called `getFile` that will return it.

The test program that was used to demonstrate the `FileTree` control earlier in this chapter contains some typical code that shows how to handle file selections in the tree. You can try this code out by running the test program again using a command like this:

```
java JFCBook.Chapter10.FileTreeTest C:\
```

and clicking on file names in the tree. Here is what the relevant piece of code looks like:

```
final FileTree ft = new FileTree(args[0]);

ft.addTreeSelectionListener(new TreeSelectionListener() {
  public void valueChanged(TreeSelectionEvent evt) {
    TreePath path = evt.getPath();
```

```
    String name = ft.getPathName(path);
    File file = ft.getFile(path);
    System.out.println("File " + name + " has been "
      + (evt.isAddedPath() ? "selected" : "deselected"));
    System.out.println("File object is " + file);
  }
});
```

Having created the `FileTree` with its root at the location specified on the command line, a `TreeSelectionListener` is added to it. When the event occurs, the `getPath` method is used to extract the first path affected by the event. In this simple example, only one path will be handled—in the general case, you might need, if you allow it, to handle more than one simultaneous selection. The `getPathName` method of `FileTree` is invoked on the path to get the full path name of the selected file and the `getFile` method is called to demonstrate that the `FileTree` can also return a `File` object for the selected node. This is all you need to know about `FileTree` to make use of it for selecting files. Here is some typical output from this program:

```
File C:\WINDOWS has been selected
File object is C:\WINDOWS
File C:\RECYCLED has been selected
File object is C:\RECYCLED
File C:\Program Files has been selected
File object is C:\Program Files
```

Custom Tree Rendering and Editing

While the general appearance of the tree control is probably sufficient for many applications, it is possible to customize several aspects of the way in which it is rendered using various techniques that will be covered in this section. Some of these techniques work no matter which look-and-feel is installed, while others need to be tailored to work differently with different look-and-feel implementations.

As well as changing a tree's appearance, it is also possible to allow the user to edit the data represented by individual tree nodes. This topic is also covered in this section.

Changing the Appearance of a Tree's Contents

There are two ways in which you can change the appearance of the nodes of a particular tree. The most flexible option is to install your own renderer that implements the `TreeCellRenderer` interface. If you do this, you have complete control over how each node is displayed. If you just want to change the appearance of the tree slightly, the simplest thing to do is just to customize the basic look-and-feel implementation. The techniques that will be shown here can be used if you want to change the appearance of one particular tree in your application. If you want to make a change that applies to all trees, you may be able to use the simpler technique that will be described in "Changing Properties for All Trees."

Customizing the Default Tree Cell Renderer

The default tree cell renderer is provided by the class `DefaultTreeCell-Renderer`, which resides in the Swing `javax.swing.tree` package. You can get a reference to the renderer that is installed in a tree by using the `JTree getCellRenderer` method and you should verify that this returns a `DefaultTreeCellRenderer`, then cast it and use the methods shown in Table 10-5 to change its behavior as required. The method works for all of the current look-and-feel implementations, because their tree cell renderers are all derived from `DefaultTreeCellRenderer`.

Table 10-5 Default Tree Cell Renderer Customization Methods

Method	Description
`public void setBackgroundNonSelectionColor(Color)`	Sets the background color used when the node is not selected.

`public void setBackgroundSelectionColor(Color)`	Sets the color in which to draw the background of a selected node.
`public void setBorderSelectionColor(Color)`	Sets the color in which to draw the border of a selected node. There is no method to set the color of the border of a nonselected node because nonselected nodes do not have borders.
`public void setTextNonSelectionColor(Color)`	Determines the color of the text when the node is not selected.
`public void setTextSelectionColor(Color)`	Determines the color of the text when the node is selected.
`public void setFont(Font)`	Sets the font used to render the node's text.
`public void setClosedIcon(Icon)`	Sets the icon displayed when the node's children are not shown.
`public void setOpenIcon(Icon)`	Sets the icon displayed when the node's children are shown.
`public void setLeafIcon(Icon)`	Changes the icon displayed when the node is a leaf.

Since each tree uses a single renderer to draw every node, changes you make using these methods are effective for the whole tree. Since a different instance of the renderer is used for each tree, changes made in this way apply only to the single tree whose renderer is modified.

For the purposes of illustration, the next example changes the background of the tree to black and uses green text for nonselected items and white text for those that are selected. The background color for selected items will also be changed to gray. The starting point for this tree is `TreeExample1`, which was created earlier. This tree shows the crews of the Skylab space station and the Apollo lunar landing missions. The code that we added to the `main` method of the program, which we renamed `ModifyRenderer`, is shown in Listing 10-4.

Listing 10-4 Customizing the default tree cell renderer

```
// Get the tree's cell renderer. If it is a default
// cell renderer, customize it.
TreeCellRenderer cr = t.getCellRenderer();
if (cr instanceof DefaultTreeCellRenderer) {
  DefaultTreeCellRenderer dtcr =
              (DefaultTreeCellRenderer)cr;
```

```
  // Set the various colors
  dtcr.setBackgroundNonSelectionColor(Color.black);
  dtcr.setBackgroundSelectionColor(Color.gray);
  dtcr.setTextSelectionColor(Color.white);
  dtcr.setTextNonSelectionColor(Color.green);

  // Finally, set the tree's background color
  t.setBackground(Color.black);
}
```

You can run this example for yourself using the command:

```
java JFCBook.Chapter10.ModifyRenderer
```

The result. If you partly expand the tree, will look something like Figure 10-10.

Figure 10-10 Customizing the colors of the nodes.

If you look at the code, you'll see that it gets the tree's renderer and checks that it is the renderer from the basic package, or one derived from it. If it is, then the text and background colors are changed. Once these changes are made, all of the nodes in the tree will be drawn with the new settings. This is a simple way to make basic changes to the tree's appearance. To make more radical changes, you need to customize the renderer that draws each node's representation.

Implementing a Customized Renderer

Like the combo box and list controls, each cell of a tree is drawn by a renderer. Tree renderers implement the `TreeCellRenderer` interface, which has only has one method:

```
public abstract Component getTreeCellRendererComponent(
    JTree tree, Object value, boolean selected,
    boolean expanded, boolean leaf, int row,
    boolean hasFocus)
```

In most cases, the most important argument passed by this method is the `value`, which is the object to be formatted. The renderer is responsible for configuring a `Component` that represents this value in the tree. In terms of the Metal look-and-feel, this component will take the place of the folder or leaf icon and the descriptive text that appears to the right of it. The remaining arguments may be used to affect the way in which the component is rendered. For example, if the node is expanded, the Metal look-and-feel shows an open folder; if it is a leaf, it displays a sheet of paper. Selected rows usually have a background of a different color, while a row with the focus might be drawn with a colored line around the border.

You've just seen that you can use public methods supplied by `Default-TreeCellRenderer` to customize it to some extent. Another way to change the way a node is rendered is to subclass the default renderer. The default renderer is actually a subclass of `JLabel` that also implements the `Tree-CellRenderer` interface. The `DefaultTreeCellRenderer` just places the text and icon it is given on the label and, aside from some work that takes place in the `paint` method to paint the background and border of the cell, most of the rendering code is inherited from `JLabel`. If you subclass `DefaultTreeCellRenderer`, you can do such things as changing the color in which the text is rendered by setting the label's attributes directly. The advantage of this is that you can vary these attributes for each individual node if necessary, rather than setting constant values that apply to the whole tree.

The following example illustrates how to make the renderer node-specific by using a different text color and icon for leaf nodes. This example is actually an extension of Listing 10-4, which works by creating and installing a subclass of the default renderer. The code that is changed is highlighted in bold in Listing 10-5.

Listing 10-5 Installing a customized version of the default tree cell renderer

```
TreeCellRenderer cr = t.getCellRenderer();
CustomDefaultRenderer dtcr = new CustomDefaultRenderer(); t.setCellRenderer(dtcr);
// Set the various colors
dtcr.setBackgroundNonSelectionColor(Color.black);
dtcr.setBackgroundSelectionColor(Color.gray);
dtcr.setTextSelectionColor(Color.white);
dtcr.setTextNonSelectionColor(Color.green);
// Set the extra attributes for this renderer
```

```
dtcr.setLeafForeground(Color.yellow);
dtcr.setLeafIcon(new ImageIcon(
SubClassRenderer.class.getResource(
      "images/earth.gif")));

// Change the row height
t.setRowHeight(0);

// Finally, set the tree's background color
t.setBackground(Color.black);
```

In this example, an instance of the new renderer is instantiated and installed using the `JTree setCellRenderer` method. The customizations that were used in the previous example can still be applied to it because it is derived from the default renderer, but these changes affect every node in the tree. One extra attribute of the custom renderer allows the color used for leaf nodes to be different from that of branch nodes. This attribute is set in the code shown above using the `setLeafForeground` method which is implemented by the custom renderer. Note also that the `setLeafIcon` method, inherited from `DefaultTreeCellRenderer`, is used to install a different icon that will be used only for leaf nodes. The only other change shown here is to set the cell row height using the `JTree setRowHeight` method, which requires an integer argument. If you supply a positive value, the rows will all be of that height, whereas a zero or negative value causes the tree look-and-feel class to query the renderer for the height of each row as it draws it, which allows variable height rows. This is necessary in this case because the icon used for leafs nodes is not necessarily the same size as that for branch nodes, so the height of each row needs to depend on the requirements of the node that will occupy it.

Core Note

The default row height is determined by the selected look-and-feel. The Metal look-and-feel does not supply a default height, so each row is as high as its renderer makes it. The Motif look-and-feel sets the default row height to 18 pixels and Windows uses 16 pixels as the default.

The custom renderer itself is actually very simple—its only job is to change the way that the `JLabel` displays its content by manipulating its properties if the node being rendered is a leaf node. The renderer is implemented here as an inner class, but you could make it equally well a freestand-ing object that can be reused. The code is shown in Listing 10-6.

Listing 10-6 A customized tree cell renderer

```
// This class is a custom renderer based
// on DefaultTreeCellRenderer
protected static class CustomDefaultRenderer
                   extends DefaultTreeCellRenderer {
  public Component getTreeCellRendererComponent(
              JTree tree,
              Object value,
              boolean selected,
              boolean expanded,
              boolean leaf,
              int row,
              boolean hasFocus) {
  // Allow the original renderer to set up the label
   Component c = super.getTreeCellRendererComponent(
                  tree, value, selected,
                  expanded, leaf, row,
                  hasFocus);

   if (leaf) {
     // Use a different foreground
     // color for leaf nodes.
     if (leafForeground != null) {
      c.setForeground(leafForeground);
     }
   }

   return c;
  }
   public void setLeafForeground(Color color) {
     this.leafForeground = color;
   }

   private Color leafForeground;
```

```
}
```

Since the intention is just to change the way in which the label created by the default renderer is configured, the first step in the `getTreeCellRendererComponent` method is to invoke the corresponding method of the default renderer superclass and let it set up the label as it would normally appear. This arranges for the appropriate text and icon to be installed, based on whether the node is a leaf or not, whether the node is selected, whether it has the focus, and whether it is expanded. Then, if the node being rendered is a leaf, the foreground text color is set to that configured by the application. If this attribute is `null`, no change is made to the value set by the default renderer. Note that the `getTreeCellRendererComponent` method determines whether the node is a leaf by using the `leaf` argument that is supplied to it. If specific behavior were required for expanded nodes or nodes that had the focus, this would also be handled by checking the method argumentnts.

To try this version of the example for yourself, use the command:

```
java JFCBook.Chapter10.SubClassRenderer
```

Figure 10-11 shows a typical snapshot of this example after some of the leaf nodes have been exposed. You'll notice that the branch nodes are rendered as they were in Figure 10-10, but the leaf nodes have an icon that shows the earth as viewed from the moon and the text is yellow.

Figure 10-11 Changing text and icon positions using a custom tree renderer.

One problem with creating a new renderer by subclassing `DefaultTree-CellRenderer` is that it may only work properly if the Metal or Windows look-and-feel is installed. Because of the fact that this example uses the original renderer functionality by invoking the `getTreeCellRendererComponent` method of its superclass, there might be a problem if it were replacing the Motif look-and-feel, which has its own cell renderer that is derived from `DefaultTreeCellRenderer`, because we wouldn't be using its specific `getTreeCellRendererComponent` method. Whether or not this matters depends entirely on what your customization is. You could, of course, implement a subclass for each of the look-and-feels and install the correct one at run time. There is, however, another technique that avoids having to generate more classes that you'll see in "Rendering and Editing Custom Objects." If you can't achieve the effects you need by subclassing the default renderer, you can provide your own by implementing the `TreeCellRenderer` interface yourself. The technique for doing this is identical to the one used when creating renderers for the `JList` and `JComboBox` control, so it isn't repeated here.

ToolTips and Renderers

By default, `JTree` does not supply tooltips, but it does support the provision of tooltips via its renderer. If you want your tree to display tooltips, you have to do two things:

- Register the tree with the `ToolTipManager`.
- Arrange for the tree's renderer to return a tooltip as outlined here.

There are three ways to handle tree tooltips.

Using the Default Renderer

The simplest approach is just to have the tree return a single tooltip that applies to the entire tree. You can achieve this very easily if the tree uses a renderer that is actually subclassed from `JLabel`, which is true of the standard renderers. All you need to do in this case is register the tree with the tooltip manager and install the tooltip as a property of the renderer itself. Registering the tree with the `ToolTipManager` can be done using the following code, where the variable `t` is assumed to hold a reference to the tree:

```
ToolTipManager.sharedInstance().registerComponent(t);
```

and setting the tooltip is equally simple:

```
((JLabel)t.getCellRenderer()).setToolTipText(
                  "Apollo and Skylab missions");
```

You can verify that this works by typing the command:

```
java JFCBook.Chapter10.ToolTipExample1
```

If you move the mouse over the tree, after a short time the tooltip will appear.

Subclassing JTree

One problem with the approach just shown is that the tooltip appears only when the mouse is over a part of the tree drawn by a renderer. If this is not sufficient, you can subclass the `JTree` class to return a tooltip whenever the mouse is over it by overriding the `getToolTipText` method and using the `JComponent setToolTipText` method to store the tooltip that you want to display. When you take this approach, you don't need to register the tree with the `ToolTipManager` because this is done by the `setToolTipText` method: The appropriate implementation of the `getToolTipText` method is as follows:

```
public String getToolTipText(MouseEvent evt) {
  return getToolTipText();
}
```

This code probably looks a little strange! In fact, `JComponent` has two overloaded variants of `getToolTipText`:

```
public String getToolTipText();
public String getToolTipText(MouseEvent evt);
```

The first of these two returns whatever string has been stored by the `set-ToolTipText` method. The second one allows a component to return a different tooltip based on the position of the mouse and is the one that is actually called by the `ToolTipManager`. The default implementation of this method in `JComponent` simply calls the other `getToolTipText` method, which is what our implementation does. The reason that we have to supply this code is that `JTree` itself overrides the second `getToolTipText` method to allow the tree's renderer to supply the tooltip, a fact that we used above (and will use again below) and, for this solution, we want to suppress `JTree`'s code and revert to that in `JComponent`. You can try this code out using the command

```
java JFCBook.Chapter10.ToolTipsExample2
```

This time, you'll find that the tooltip appears when the mouse is over any part of the tree.

Node-specific Tooltips Using a Custom Renderer

A third approach to the problem, and the most powerful, is to implement tooltip support in the renderer. This makes it possible to return a different tooltip for each node if necessary. As we said in the previous section, `JTree` overrides the `getToolTipText` method to get the tooltip for each node from that node's renderer. It does this by calling the renderer's `getTree-CellRendererComponent` method as if it were going to render the node and then invokes the `getToolTipText` method of the returned component to get the tooltip for the node. If you want your nodes to display tooltips, you simply have to call `setToolTipText` on the component that you return from the renderer's `getTreeCellRendererComponent` method.

To demonstrate the general technique, we'll modify the custom renderer shown in Listing 10-6 to include a tooltip. For simplicity, we'll make the tool-tip be the same text as would be displayed on the node itself. In a real-world application, you would probably store the tooltip as part of the node or its user object and code the renderer to extract it from there. Setting the tooltip requires only one extra line of code in the renderer's `getTreeCellRen-dererComponent` method:

```
// By default, use the same value as the
// node text for the tooltip, for leaf nodes only
 ((JComponent)c).setToolTipText(
            leaf ? value.toString() : null);
```

where the variable `c` is the component being returned by the renderer. In this example, we set a tooltip only for leaf nodes.

Having made this change, you just need to register the tree with the Tool-TipManager, as shown earlier in this section, and your tree will display a different tooltip for each leaf node. However, as with the first approach we took to displaying tooltips, a tooltip will only be displayed when the mouse is over an area that will be drawn by a renderer. To arrange for a tooltip to be displayed wherever the mouse is, we can combine this approach with the previous one by subclassing `JTree` and implementing a cleverer version of

getToolTipText:

```
public String getToolTipText(MouseEvent evt) {
  // Let the renderer supply the tooltip
  String tip = super.getToolTipText(evt);

  // If it did not, return the tree's tip
  return tip != null ? tip : getToolTipText();
}
```

This code first calls the `JTree` implementation of `getToolTipText`, which allows the renderer to return a tooltip. If it does not supply one, then the tool tip set using `setToolTipText` is returned instead. Now, you can set a global tooltip using the `JTree setToolTipText` method and it will appear whenever the mouse is over a part of the tree that is not drawn by a renderer, or when the renderer itself returns a `null` tooltip which, in this case, would be for non-leaf nodes. You can try this out using the command

```
java JFCBook.Chapter10.ToolTipsExample3
```

You'll find that a generic tooltip is displayed when the mouse is not over a leaf node and that each leaf node displays its own text as a tooltip. If you take this approach, you must register the tree with the `ToolTipManager` unless you invoke `setToolTipText`, in which case registration is handled for you.

Changing the Basic Look-and-Feel Specific Parts of a Tree

If your tree uses a look-and-feel that is derived from `BasicTreeUI`, you can customize more of its appearance. You can find out at runtime whether this is the case by getting a reference to its UI class by calling the `JComponent getUI` method and checking whether it is an instance of `BasicTreeUI`. If it is, as it will be for all of the standard look-and-feel implementations, you can make use of the methods shown in Table 10-6 to change the way in which the framework of the tree is drawn.

Table 10-6 Basic Tree Look-and-Feel Customization Methods

Method	Description
`public void setLeftChildIndent(int)`	Sets the horizontal distance between the left side of a node and the expansion handle for its children.
`public void setRightChildIndent(int)`	Sets the horizontal distance between the center of a node's expansion handle and the left side of its rendered cell.
`public void setCollapsedIcon(Icon)`	Sets the icon used to render the expansion handle when the node's children are not shown.
`public void setExpandedI-con(Icon)`	Changes the icon used as the expansion icon handle the node's children are on view.

Figure 10-12 shows the various configurable parts and how they affect the way in which the tree is rendered.

Figure 10-12 Basic Tree UI Customizable Items

These settings are applied everywhere in the tree—you can't have different settings for different rows. The icons that you supply are used to represent the expansion handle that appears to the left of a node that has children. Normally, this icon is a small circle with a "handle" extension or small square with a plus sign (+) in it when the node is collapsed and a minus sign (-) when it is expanded. The default icons are all fairly small. If you substitute larger ones, you will need to change the right or left indent values to space the remaining elements out horizontally. If your icon is taller than the standard ones, you'll need to increase the tree's row size, which you can do with the `setRowHeight` method of `JTree`. The next example adds these customizations to the tree that we have been using throughout this chapter. New expansion icons have been substituted and, since they are larger then the standard icons, the right child indentation and the row height have been increased to allow space for them. You can run this example for yourself using the command:

```
java JFCBook.Chapter10.CustomTree
```

The result will look like that shown in Figure 10-13

Figure 10-13 A tree with customized expansion icons.

Here's the code that was added to the `main` method of the previous example, immediately after creating the tree, to make these changes. In this code, `t` holds a reference to the `JTree` object:

```
// Get the tree's UI class. If it is a BasicTreeUI,
// customize the icons
ComponentUI ui = t.getUI();
 if (ui instanceof BasicTreeUI) {
  ((BasicTreeUI)ui).setRightChildIndent(21);
  URL expandedURL = CustomTree.class.getResource(
                                "images/closebutton.gif");
  URL collapsedURL = CustomTree.class.getResource(
                                "images/openbutton.gif");
  ((BasicTreeUI)ui).setExpandedIcon(new ImageIcon(expandedURL));
  ((BasicTreeUI)ui).setCollapsedIcon(new ImageIcon(collapsedURL));
}
```

Since these customizations only work for a look-and-feel that bases its tree UI on `BasicTreeUI`, the first step is to use `getUI` to get the UI class. If it turns out that it is actually a `BasicTreeUI`, then new icons are loaded using the familiar `ImageIcon` class and used to customize the expansion handles. Note that extra space has been added between the expansion handle and the node itself to account for the fact that the customized icons for the expansion handles are larger than the default ones.

Changing Properties for All Trees

If you want to change the color of certain parts of a tree or the icons that it uses to represent open and closed folders and leaf nodes, and you want this change to apply to every tree in your application, you don't need to implement your own renderer or modify an existing one—instead, you can modify the `UIDefaults` table for the current look-and-feel, which maps property names to the values appropriate for the current look-and-feel. The tree has 19 useful properties that you can set in this way. These properties, together with the types of the associated objects, are listed in Table 10-7. The `UIDe-faults` table and the associated `UIManager` class are discussed in detail in Chapter 13.

Table 10-7 UIDefaults Properties for JTree

Property Name	Object Type
Tree.background	Color

Tree.closedIcon	Icon
Tree.collapsedIcon	Icon
Tree.editorBorder	Border
Tree.expandedIcon	Icon
Tree.font	Font
Tree.foreground	Color
Tree.hash	Color
Tree.leafIcon	Icon
Tree.leftChildIndent	Integer
Tree.openIcon	Icon
Tree.rightChildIndent	Integer
Tree.rowHeight	Integer
Tree.scrollsOnExpand	Boolean
Tree.selectionBackground	Color
Tree.selectionBorderColor	Color
Tree.selectionForeground	Color
Tree.textForeground	Color
Tree.textBackground	Color

If you want to change an entry in this table, there are two possible approaches depending on the effect you want to achieve. If you need the changes to stay in place if the user dynamically switches the look-and-feel (if you allow this—see Chapter 13), you should store `Color`, `Font`, `Integer`, `Boolean`, `Border`, or `Icon` objects in the `UIDefaults` table. When a look-and-feel switch occurs, these values will not be changed.

On the other hand, if the attributes you are setting only work with the current look-and-feel, you need to allow them to be reset to the new look-and-feel's defaults when a look-and-feel switch occurs. To achieve this, you need to wrap the attribute you are changing with an instance of `BorderUIRe-source` for a border, FontUIResource for a font, `IconUIResource` in the case of an icon, or `ColorUIResource` for a color. For example, to change the color of the text for a selected node to yellow only while the current look-and-feel is selected, you would proceed as follows:

```
UIDefaults defaults = UIManager.getDefaults();
defaults.put("Tree.selectionForeground",
          new ColorUIResource(Color.yellow));
```

or to change the color of the lines that connect the nodes to white, you might do this:

```
UIDefaults defaults = UIManager.getDefaults();
defaults.put("Tree.hash",
          new ColorUIResource(Color.white));
```

On the other hand, the following lines makes these changes permanent:

```
UIDefaults defaults = UIManager.getDefaults();
defaults.put("Tree.selectionForeground", Color.yellow);
defaults.put("Tree.hash", Color.white);
```

You can find out more about the `UIDefaults` table in Chapter 13. In general, it is a good idea to check whether you can carry out customization by modifying this table before using a more complex approach.

Editing a Tree's Nodes

By default, trees are read-only to the user but, if you wish, you can allow the user to change the value associated with any node in the tree with just one line of code:

```
tree.setEditable(true);
```

If a tree is editable, you can edit the nodes by clicking on a node three times in quick succession or clicking once to select the node and then clicking again, having paused for a short time so that the two clicks do not get merged into a double-click (note that once a node is selected, a single click is enough to start the editing process), or by selecting a node and pressing the `F2` key. Any of these gestures will causes the node's editor to appear. By default, the editor is a text field displaying the same text that represents the node in the tree. To edit the node, you just change the text and press `RETURN`. The editor will disappear and the new value will be installed.

To show how editing works, let's look at yet another version of the hardworking example program that we have used throughout this chapter, this time called `EditableTree`. The change to make it editable is just one line of code but, so that you can see what happened, a `TreeModelListener` is attached to the tree. When the editor accepts a change, it applies it to the tree model, which causes a `TreeModelEvent`. The code that was added to display the content of this event is shown below:

```
t.setEditable(true);

t.getModel().addTreeModelListener(
          new TreeModelListener() {
  public void treeNodesChanged(TreeModelEvent evt) {
    System.out.println("Tree Nodes Changed Event");
    Object[] children = evt.getChildren();
    int[] childIndices = evt.getChildIndices();
    for (int i = 0; i < children.length; i++) {
      System.out.println("Index " + childIndices[i] +
        ", changed value: " + children[0]);
     }
  }
```

```
  public void treeStructureChanged(TreeModelEvent evt) {
    System.out.println("Tree Structure Changed Event");
  }
  public void treeNodesInserted(TreeModelEvent evt) {
    System.out.println("Tree Nodes Inserted Event");
  }
  public void treeNodesRemoved(TreeModelEvent evt) {
    System.out.println("Tree Nodes Removed Event");
  }
});
```

The event that the listener gets will indicate that one node has changed, so of the listener methods shown above, only `treeNodesChanged` will ever be called, but you are obliged to provide the others to implement the listener interface. The event handler extracts the list of changed indices from the event and the array of changed objects. Code is included to print a set of these but, in fact, there will only ever be one. The object in the array returned by `getChildren` is the `DefaultMutableTreeNode` for the node that was edited, so printing it will cause its `toString` method to be called that, as you know, will print the text associated with the node. If you run this program using the command

```
java JFCBook.Chapter10.EditableTree
```

and change the text "Apollo" to "Shuttle," you'll see the following event generated:

```
Tree Nodes Changed Event
Index 0, changed value: Shuttle
```

To edit the tree cell, click the mouse over the text of value you want to change, then pause briefly and click again. If your clicks are too close together, nothing happens in the case of a leaf node, but if it's a branch node, you'll end up expanding or collapsing it instead of editing it. If you accidentally start editing a cell and don't want to continue, click anywhere else in the control or press the `ESCAPE` key and the edit will be aborted.

Custom Cell Editors

If you allow your tree to be edited, it is up to you to carry out whatever action is implied by the change and to check that the new value is legal. One way to make it easier to get legal input values is to supply a custom editor that only offers the legal values, such as a `JComboBox`. A custom editor must implement the `TreeCellEditor` interface:

```
public interface TreeCellEditor extends CellEditor {
  public Component getTreeCellEditorComponent(JTree tree,
    Object value, boolean isSelected, boolean expanded,
    boolean leaf, int row);
}
```

The `TreeCellEditor` interface extends `CellEditor`, which contains methods that are also used to create editable tables, as you'll see later in the next chapter. The `getTreeCellEditorComponent` method returns a component that can edit a value in the tree; the initial value is passed as the second argument, along with three arguments that specify whether the node being edited is selected or expanded and whether it is a leaf. The last argument gives the screen row of the node. There is only one `TreeCellEditor` for each tree. As the user edits different nodes, this method is called with different values and may, if necessary, return a differently configured component. Typically, the appearance of the returned component depends on the three boolean arguments, because it occupies the same space in the tree as a cell renderer and therefore may need to draw an appropriate icon (e.g., an open or closed folder).

The methods shared with editors and used with `JTable` (see Chapter 11, "The Table Control") are as follows:

```
public interface CellEditor {
    public Object getCellEditorValue();
    public boolean isCellEditable(EventObject);
    public boolean shouldSelectCell(EventObject);
    public boolean stopCellEditing();
    public void cancelCellEditing();
    public void addCellEditorListener( CellEditorListener);
    public void removeCellEditorListener( CellEditorListener);
}
```

These methods all deal with the mechanics of setting up an editor and starting and stopping the editing process. You don't need to implement all seven of these methods to change the text-based node editor,

because most of the work is done for you in the `DefaultCellEditor` class that provides a complete implementation of an editor that can appear as either a text field, a checkbox, or a combo box. Here are its three constructors:

```
public DefaultCellEditor(JTextField text);
public DefaultCellEditor(JCheckBox box);
public DefaultCellEditor(JComboBox combo);
```

To install a custom editor in a tree, you need to first create the editor (usually by using or subclassing `DefaultCellEditor`), wrap it in an instance of the `DefaultTreeCellEditor` class, and then finally call the `JTree` `setCellEditor` method. Two classes are used rather than one because `DefaultCellEditor` is a generic class that can provide editors suitable for use with both `JTree` or `JTable`, while `DefaultTreeCellEditor` augments any `CellEditor` to provide the semantics appropriate for editing in a tree.

To add a combo-box editor to a tree requires only a few lines of code. The `ComboTree` example, that you can run with the command:

```
java JFCBook.Chapter10.ComboTree
```

was created by adding the following to the EditableTree program:

```
t.setEditable(true);

final String[] values = { "Mercury", "Gemini", "Apollo",
           "Skylab", "Shuttle" };

JComboBox combo = new JComboBox();
for (int i = 0 ; i < values.length; i++) {
  combo.addItem(values[i]);
}

DefaultTreeCellEditor editor = new DefaultTreeCellEditor(
              t, dtcr, new DefaultCellEditor(combo));

t.setCellEditor(editor);

t.getModel().addTreeModelListener(
               new TreeModelListener() {
```

Because the `DefaultCellEditor` and `DefaultTreeCellEditor` classes do all of the work, all you need to do is populate the combo box that will appear in the tree and then create the `DefaultCellEditor` and `DefaultTreeCellEditor` classes that will manage it. The default cell editor is replaced by calling the `JTree` method `setCellEditor`. If you run this example and use one of the standard editing gestures on the `Apollo` node, the combo box appears, as shown in Figure 10-14. Click on the arrow and the list of possible values drops down. When you select one, the drop-down closes and the editor disappears. As with the text editor, selecting a new value causes a `TreeModelEvent` that contains the new value.

The result will be something like that shown in Figure 10-14. The `DefaultTreeCellEditor` class has two constructors:

```
public DefaultTreeCellEditor(JTree tree,
              DefaultTreeCellRenderer renderer)
public DefaultTreeCellEditor(JTree tree,
              DefaultTreeCellRenderer renderer,
              TreeCellEditor editor)
```

Figure 10-14 A tree with a Combo Box cell editor

The first constructor installs a new editor that is based on a text field and is the one used when the tree installs its own default editor. If you want to use a custom editor, you will need to use the second constructor and pass your editor as the third argument, as shown in the code extract above. The second argument of both constructors is a `DefaultTreeCellRenderer`—why is this required when installing a custom editor? When editing starts, the tree will add the editing component to the tree in place of the tree's node. In most cases, however, you would not want to lose the node's icon if it has one. To make sure that the rest of the node is displayed as it is when there is no edit in progress, you need to pass a reference to the tree's renderer to the constructor. If you supply `null`, the editor will still be installed but the icon will disappear until the edit is completed.

Controlling Which Nodes Can Be Edited

This simple example has a major flaw. As noted earlier, there is one editor that is shared among all of the nodes in the tree. This being the case, run the previous example again and open the tree to show the nodes for the crew of, say, Apollo 11. As you do this, be careful to click only on the expansion icons

and not on the nodes themselves or on the text, to avoid initiating the editor. Now click once on the name of any crew member and press `F2`. In response, you'll get the same combo box offering a choice of spacecraft! This is clearly not appropriate.

The problem here is that we have made the tree editable by calling `setE-ditable(true)`, which allows all of its nodes to be edited. What we need is to be able to specify editability at a lower level of granularity, so that we can stop the editor appearing for specific nodes or types of node. There are two ways to do this, one of which involves subclassing `JTree`, the other being implemented in the tree cell editor.

Controlling Editability by Subclassing JTree

When the user makes one of the gestures that initiates editing, the following `JTree` method is called:

```
public boolean isPathEditable(TreePath path);
```

where the `TreePath` corresponds to the node to be edited. The default implementation of this method looks like this:

```
public boolean isPathEditable(TreePath path) {
  return isEditable();
}
```

As a result of this, any cell can be edited provided that the tree itself is editable (although the editor can still refuse to allow the edit as you'll see below). One way to gain control of the editing mechanism is to override this method and apply your own criterion to determine whether the given cell should be edited. As an example, to arrange that only leaf nodes can be edited, you could implement the `isPathEditable` method as follows:

```
public boolean isPathEditable(TreePath path) {
  if (isEditable()) {
    return getModel().isLeaf(
          path.getLastPathComponent());
  }
  return false;
}
```

Note that it is necessary to ensure that the tree is editable by calling the `isEditable` method before applying a more specific test to the `TreePath` itself.

Controlling Editability in the Tree Cell Editor

Even if the tree is editable and the selected path is editable according to the `isPathEditable` method, it is still possible to stop the user editing a node by overriding the `isCellEditable` method of the `CellEditor` implementation. If this method returns `false`, no editor will be displayed. The `isCellEditable` method is defined as follows:

```
public boolean isCellEditable(EventObject evt)
```

The only argument that this method receives is an `EventObject`. The meaning of this argument depends on how this method is invoked:

- When the user clicks on the node with the mouse, this argument is the `MouseEvent`that was delivered to the tree. The coordinates of the event correspond to some location within the rendered area of the node.
- If the user attempts to start an edit using the `F2`key or programmatically (see below), this argument is `null`.
- If the edit is started because the user clicked in a cell that was already selected, this argument is again passed as `null`.

Usually, the `isCellEditable` method is used to allow or disallow edits based on the way in which the edit is started—for example, the editor may allow any edit started via the keyboard (provided the `isPathEditable` method has already allowed it), but only allow an edit initiated using the mouse on a triple click, to determine which it would inspect the `Mou-seEvent`. This method is not usually used to make a decision based on which node is being edited and it is difficult to do so, because the affected node is not passed as a method argument. Granted, you can locate the affected node by calling the `JTree getPathForLocation` method using the coordinates in the `MouseEvent`, but this is only possible in one of the three possible cases for which this method might be called.

Nevertheless, it is sometimes useful to make node-dependent decisions in the editor. To do this, you have to store a reference to the node being edited when the `getTreeCellEditorComponent` method is called, which happens just before `isCellEditable` is invoked. An example implementation for our tree example is shown below.

```
DefaultCellEditor comboEditor =
         new DefaultCellEditor(combo);
DefaultTreeCellEditor editor = new DefaultTreeCellEditor(
                      t, dtcr, comboEditor) {
  private Object lastValue;
    public Component getTreeCellEditorComponent(JTree tree,
        Object value, boolean isSelected, boolean expanded,
        boolean leaf, int row) {
      // Store the path
      lastValue = value;
      return super.getTreeCellEditorComponent(tree, value,
               isSelected, expanded, leaf, row);
  }

  public boolean isCellEditable(EventObject evt) {
    if (super.isCellEditable(evt) &&
        lastValue instanceof DefaultMutableTreeNode) {
      Object userObject =
        ((DefaultMutableTreeNode)lastValue).
                     getUserObject();
       for (int i = 0; i < values.length; i++) {
        if (userObject.equals(values[i])) {
          return true;
        }
      }
    }
    return false;
  }

};

t.setCellEditor(editor);
```

This example can be run with the command:

```
java JFCBook.Chapter10.ComboTree2
```

If you select several nodes in the tree in turn and try to edit them, you'll find that you are prevented from doing so, unless you select one of the spacecraft nodes (initially labeled "Apollo" and "Skylab"). The code itself is straightforward given the discussion that preceded it. The `getTreeCellEd-itorComponent` simply stores the value in the tree that it is given to edit and then calls the superclass implementation to get the actual editor. The `value` argument passed to this method is actually the last component of the `Tree-Path` for the node being edited, which will be a `DefaultMutableTreeNode` in this case.

The overridden `isEditable` method first ensures that the tree itself agrees to the current node being edited by calling the superclass method that it overrides. Assuming that this is the case, it retrieves the `DefaultMu-tableTreeNode` stored by the `getTreeCellEditorComponent` method and extracts its user object. The problem is how to determine that the node that the user wants to edit corresponds to a spacecraft. In this case, the approach taken is to compare the user object with all of the values stored in the combo box, which represent all of the legal values that this node could hold. This will work for this specific example. Another possible approach would be to define the tree nodes in such a way that they include an attribute that indicates whether they can be edited, or to implement specific tree node classes that represent spacecraft and astronauts and perform the check by inspecting the runtime type of the node.

Programmatic Control of Editors

Most tree editing will be initiated by the user. The user interface gestures that initiate and end the editing process are detected by the tree's UI delegate class which listens to mouse events and installs keyboard mappings for the F2 key and any other keys that a custom look-and-feel might want to use for editing. The UI class makes use of a relatively small API provided by JTree that actually initiates and terminates edits. The same API can be used by application code to start or end an editing session in response to different user gestures or as a result of some state change in the application. The methods that you can use are defined in Table 10-8.

Rendering and Editing Custom Objects

The replacement editor developed in the previous section had an implicit assumption that doesn't hold for some trees. The assumption was that the user objects associated with the tree's nodes were all strings, which was (conveniently) true in that particular example. There is no reason for this always to be true and in general it is useful to be able to associate an arbitrary object with a node.

When the user object is not a string, the default renderer will display it properly provided that it provides a suitable `toString` method because, as you already know, the renderer applies this method to the node, which will in turn apply it to the user object. However, editing such an object is not so simple. Let's return to the combo box editor to see what the issue is. When the tree was built for this case, all the user objects were strings and the combo box was populated with all of the possible string values that would be meaningful for the nodes that could be edited. When the user selects a new value, the editor applies the value to the tree. Because the editor was derived from the `DefaultTreeCellEditor` class, there was no need to supply the code

Table 10-8 Tree Editing Methods

Method	Description
public void startEditingAtPath(TreePath path)	Initiates an edit of the given path. The tree is scrolled so that the node corresponding to path is visible, if necessary and checks are made to ensure that the tree and node are both editable, as described earlier. If these conditions are not met, or there is no editor installed in the tree, the edit will not start. The `isEditing` method can be used to determine whether the edit was started.
public boolean stopEditing()	This method is used to request a tidy end to the editing session, in which the new value supplied by the user, if any, is applied to the tree. If no edit is in progress, this method simply returns true. Otherwise, the editor's `stopCellEditing` method is called. If the editor returns `false`, then the edit will continue and `stopEditing` itself will return `false`. If the editor returns `true`, its `getCellEditorValue` method is called to get the value that the user supplied and this

	value is stored in the node being edited using the `TreeModel valueForPath-Changed` method, which will be described in the next section. Finally, the editor is removed from the tree and `stopEditing` returns `true` to indicate success.
public void cancelEditing()	By contrast to `stopEditing`, this method requests that the current edit be forcibly terminated. Its operation is very similar to that of `stopEditing`, with the following differences:
	1. The editor is not asked to stop editing via its `stopCellEditing` method; instead, its `cancel-CellEditing` method is called. This requires it to unconditionally abandon its editing session.
	2. The value in the editor is not written to the tree. Thus, any changes made by the user in the editor are lost.
	This method is typically called when the user makes a gesture that signals he or she does not not want to continue with the edit, such as pressing the `ESCAPE` key or selecting a different node.
public TreePath getEditingPath()	Returns the path that is currently being edited or `null`if there is no edit in progress.
public boolean isEditing()	Returns `true`if there is an edit in progress, `false`if there is not.
public boolean isPathEditable(TreePath path)	Returns `true`if the path given by the argument is editable. The default implementation of this method returns `true`if the tree itself is editable and `false`if it is not. `Jtree`subclasses can override this method to return different values for different nodes, as

	shown earlier in this section.

that carried out this step. What happens is that the editor's `stopCellEdit-ing` method is called when the edit is finished and this method informs any registered `CellEditorListener`s that editing is complete. The tree's UI class registers as a `CellEditorListener` and, when it receives this event, it gets the new value from the editor by calling its `getCellEditorValue` method, which, in the case of our example, will return the value selected from the combo box, courtesy of code provided by `DefaultTreeCellEdi-tor`.

Core Note

`CellEditorListener` is not a very useful interface unless you are implementing a tree UI or an editor. It consists of two methods:

```
public interface CellEditorListener
           extends EventListener {
    public abstract void editingStopped(ChangeEvent evt);
    public abstract void editingCanceled(
                        ChangeEvent evt);
}
```

The `editingStopped` method is called when editing completes normally, while `editingCanceled` is invoked when editing is aborted because, for example, the user selected a different node on the tree or collapsed the branch that contained the node being edited. This interface doesn't have an event of its own—it just supplies a `ChangeEvent`, an event that has no state, but just indicates that something about the event source may have changed.

The tree's UI class stores the edited value in the model by invoking the `DefaultTreeModel valueForPathChanged` method, which is defined as follows:

```
public void valueForPathChanged(TreePath path,
                        Object value);
```

The default implementation of this method locates the `DefaultMu-tableTreeNode` for the affected path and stores the new value by invoking `setUserObject(value)`. In our earlier example, the combo box returned a `String` value, so the type of the `value` argument to this method will be `String`, which means that the user object will be replaced by its own `String` representation. Unless the user object in the node that the editor was editing was originally a `String` (which, fortunately, it was in our example), this would be a fatal mistake.

To get around this problem, you can do one of two things:

- Populate the combo box with the user objects themselves instead of strings. This works as long as the user objects provide a `toString`method that returns a `String`that is meaningful to the user, because the combo box displays the result of invoking `toString` on each of its displayed items, but the selected item would be an instance of the user object type. Therefore, `valueForPathChanged` will be passed an object of the correct type.

- Continue to populate the combo box with strings, but override the `valueForPathChanged`method to substitute an appropriate object for the `String`when the edit is complete.

The first method is the much simpler of the two and really requires no further explanation so, for the purposes of illustration, let's look at how to implement the second method.

Editing Trees with Custom User Objects

Let's start by deciding what the user object that will be associated with a node might be. Obviously, this depends on how you are going to use the tree, but one common way to create a tree is to define a user object that contains a custom icon and a text string that can be used for display purposes, as well as any other state information that might be of use to the application itself. Storing both of these attributes in the user object allows you to have a different icon for each object in the tree instead of using the default icon provided by the look-and-feel, or using a single substitute that is the same for all nodes (or at least for all leaf nodes), as you saw when looking at how to create a custom renderer earlier in this chapter. Here's the

plan for this example.

Instead of populating the tree with nodes that use `String`s as the user object, this example will invent an interface that allows an object to return an icon to be rendered to represent it, then create suitable objects that implement this interface and associate them with nodes in the tree. To correctly display these nodes, a custom renderer will be needed. This renderer will get the text and the icon from the user object instead of using built-in icons or substitute icons that don't depend on the object being displayed. To edit these nodes, an editor that offers a combo box interface will be used. To ensure that the value stored in a node after the edit has completed is an object of the correct type, it will be necessary to supply our own tree model in which the `valueForPathChanged` method is overridden.

Let's start by defining the interface:

```
// An interface that describes objects with icons
interface NodeWithIcon {
  public Icon getIcon();
}
```

From the point of view of the tree, all that is required of the user objects is the ability to get an icon and a text description from them. It isn't necessary to include a method in the interface to get a text description, because the `toString` method, which is defined for every `Object`, can be used for this.

Core Note

For simplicity, the implementation of this example is kept in a single source file. Because there can only be one public class or interface per source file, the above interface is declared with package scope. If you were developing a real application using the technique that we are explaining here, you would extract this interface into its own source file and make it public.

Next, let's define the user objects that will be used to populate the tree for our example. This example is going to use a custom data model so that a new `valueForPathChanged` method that knows how to map from a `String` to the appropriate user object that corresponds to that `String` can be implemented. In other words, the data model must be aware of the meaning of the objects that it contains, so the definition of the data objects is tied to the data model by defining them in terms of an inner class of the data model itself. In fact, the data model is going to be defined in such a way that its users need only know that the items that it contains implement the `NodeWithIcon` interface, which allows us to use other data models in place of the one that will be shown here.

Listing 10-7 shows the implementation of the tree data model and the items that will be used as the user objects in the tree.

Listing 10-7 A customized tree model class

```
// This class represents the data model for this tree
class CustomTreeModel extends DefaultTreeModel {
  public CustomTreeModel(TreeNode node) {
    super(node);
    objects.put("CSM",
         new HardwareObject("CSM", "images/csmicon.gif"));
    objects.put("LM",
         new HardwareObject("LM", "images/lemicon.gif"));
  }

  public void valueForPathChanged(TreePath path,
                                  Object newValue) {
    DefaultMutableTreeNode node =
        DefaultMutableTreeNode)path.getLastPathComponent();
    String value = (String)newValue;
    // Use the string to locate a suitable HardwareObject.
    // If there isn't one, just use the string as the
    // user object
    HardwareObject object = getObjectForName(value);
    if (object != null) {
      node.setUserObject(object);
    } else {
      node.setUserObject(value);
    }
    nodeChanged(node);
  }
```

```
// Get the HardwareObject for a given name, or
// null if unknown
public HardwareObject getObjectForName(String name) {
  return (HardwareObject)objects.get(name);
}

// Create the set of objects that can populate
// the leaf nodes
protected static class HardwareObject
               implements NodeWithIcon {
  protected HardwareObject(String text,
                           String imageName) {
    this.text = text;
    this.icon = new ImageIcon(getClass().getResource(
                              imageName));
  }

  // Return text for display
  public String toString() {
    return text;
  }

  // Return the icon
  public Icon getIcon() {
    return icon;
  }

  protected String text;
  protected Icon icon;
}

// Get the possible hardware types in a combo box
public JComboBox getNamesAsCombo() {
  DefaultComboBoxModel model =
             new DefaultComboBoxModel();
  Enumeration e = objects.keys();

  while (e.hasMoreElements()) {
    model.addElement(e.nextElement());
  }
  return new JComboBox(model);
  }
  protected static Hashtable objects = new Hashtable();
}
```

Let's look first at the user object class, which is called `HardwareObject`. This particular example will display two items of space hardware and allow the user to place either one of them into a particular location in the tree by offering a combo box with their names when the user clicks to start editing. To the left of the name, there will be a small picture that shows what the item actually looks like. To keep this example simple, only leaf items will be customized, so there is no need to supply extra icons for expanded and collapsed nodes. The same principles would, of course, be applied if you wanted to produce a completely general solution.

To create each `HardwareNode` object, the name of the hardware and the location of its icon must be supplied. The implementation of the `Hard-wareNode` class is very simple. The constructor stores the hardware name and uses the `ImageIcon` class to load the corresponding icon. The `toString` method returns the object's name and the `getIcon` method returns the icon. This method is needed because `HardwareNode` is required to implement the `NodeWithIcon` interface.

Because we don't want anything outside the data model to know how to create these user objects, two of them are built in the data model's constructor and placed in a hash table, keyed by their names. To complete the encapsulation, a method (`getObjectForName`) is provided to get a copy of the object given its name. You'll see why this method is needed later. These objects are both read-only, so it is only necessary to create one of each. Storing one copy of each in the hashtable is, therefore, acceptable, no matter how many times it might appear in the tree (and in this example it appears either once or not at all).

Thinking ahead a bit to the implementation of the cell editor, a combo box will need to be populated with the names of the possible hardware choices. Again, it isn't desirable to have the editor know how many possible choices there are or what relationship the string name that it will store might have to the objects themselves, so we implement a method within the data model (`getNamesAsCombo`) that returns a suitably populated `JComboBox` that the editor can use. This combo box is created by obtaining an enumeration of the items in the data model's internal hash table and adding one combo item per entry in the enumeration.

The valueForPathChanged Method

The only other method of the data model that still needs to be described is the one that this section opened with—`valueForPathChanged`. This method is called with a `TreePath` for the node that has been edited and the new value which will have come from the combo box and which is therefore known to be a `String`, because the combo box was created using the `get-NamesAsCombo` method, which populates it with `String`s. What `value-ForPathChanged` needs to do is find the correct user object corresponding to the `String` returned from the editor and install it in the node. Finding the user object is easy—the data model has a hashtable that maps from the name to the object itself, so all that is necessary is to invoke the `getObjectFor-Name` method, which accesses the hashtable to locate the object. This method only returns an appropriate answer if the `String` returned by the combo box was the key for one of the objects that are in the hashtable. Given the way that this example has been written, this will always be true, but if you take the code here and use it in another application which allows the combo box to be editable (probably a mistake!), the user may type a meaningless value and you won't find a corresponding object in the hashtable. To defend against this possibility, if this happens, `getObjectForName` will return `null`, and the exact value that the user typed is stored as the user object. Whether or not this is of any use depends on the application: It does at least stop the program getting a `NullPointerException`.

Having got the correct object from the `getObjectForName` method, the next step is to find the node to install it in. You already know how to do this—you just apply `getLastPathComponent` to the `TreePath` passed to `val-ueForPathChanged` to find the node and then call `setUserObject` to install the user object. Finally, because the content of the model has changed, the `nodeChanged` method must be called to have the data model send events to registered `TreeModelListener`s. This will cause the tree's visual state to be updated.

That takes care of the data model. However, this example also needs three other things:

1. A `JTree`to display the model.
2. An editor that provides a combo box with the legal hardware types installed.
3. A renderer that will display the hardware object in each of the tree's leaf nodes.

We'll cover each of these items separately. For reference, the complete implementation, apart from the data model, which was covered earlier, is shown in Listing 10-8.

Listing 10-8 Rendering and editing nodes with custom icons

```
package JFCBook.Chapter10;

import javax.swing.*;
import javax.swing.tree.*;
import javax.swing.event.*;
import java.util.*;
import java.awt.Component;
import java.awt.event.*;

public class CustomIcons {
  public static void main(String[] args) {
    JFrame f = new JFrame("Custom Icons");

    // Create the model with just the root node
    DefaultMutableTreeNode rootNode =
                  new DefaultMutableTreeNode("Apollo");

    CustomTreeModel m = new CustomTreeModel(rootNode);
    JTree t = new JTree(m) {
      public boolean isPathEditable(TreePath path) {
        // Only allow editing of leaf nodes
        return isEditable() && getModel().isLeaf(path.getLastPathComponent());
      }
    };
  t.putClientProperty("JTree.lineStyle", "Angled");
  t.setEditable(true);
  t.setRowHeight(0);// Row height is not fixed

  // Define a replacement renderer
  final TreeCellRenderer oldRenderer = t.getCellRenderer();
  DefaultTreeCellRenderer r = new DefaultTreeCellRenderer()
```

```
{
    private Object lastValue;

    public Component getTreeCellRendererComponent(
                  JTree tree, Object value,
                  boolean selected, boolean expanded,
                  boolean leaf, int row,
                  boolean hasFocus) {
      // Store the value for getLeafIcon
      lastValue = value;

      // Allow the original renderer to set up the label
      Component c =
        oldRenderer.getTreeCellRendererComponent(
                      tree, value, selected,
                      expanded, leaf,
                      row, hasFocus);
      // Now change the icon if necessary
      if (leaf && c instanceof JLabel) {
        JLabel l = (JLabel)c;
        Icon icon = getLeafIcon();
        if (icon != null) {

        l.setIcon(icon);
        }
      }

      return c; }

      public Icon getLeafIcon() {
        Object o =
        ((DefaultMutableTreeNode)lastValue).getUserObject();
        if (o instanceof NodeWithIcon) {
          return ((NodeWithIcon)o).getIcon();
      } else if (oldRenderer
              instanceof DefaultTreeCellRenderer) {
        return ((DefaultTreeCellRenderer)oldRenderer).
                      getLeafIcon();
      }
      return null;
    }
```

```
  };
  t.setCellRenderer(r);

  // Define a replacement editor
  JComboBox combo = m.getNamesAsCombo();
  DefaultCellEditor comboEditor =
         new DefaultCellEditor(combo);
  DefaultTreeCellEditor editor =
         new DefaultTreeCellEditor(t, r, comboEditor);
  t.setCellEditor(editor);

  // Populate the model with a small amount of data
  DefaultMutableTreeNode hardwareNode =
              new DefaultMutableTreeNode("Hardware");
  rootNode.add(hardwareNode);
  String itemName = (String)combo.getItemAt(0);
  DefaultMutableTreeNode itemNode =
      new DefaultMutableTreeNode(
                  m.getObjectForName(itemName));
      hardwareNode.add(itemNode);

      t.expandRow(0);
      t.expandRow(1);

      f.getContentPane().add(new JScrollPane(t));
      f.setSize(300, 300);
      f.setVisible(true);
  }
}
```

The Tree Implementation

Since the tree is just going to display the data model and allow its custom renderer and editor to handle the details of displaying and editing the nodes, almost nothing needs to be done to implement it. The only special task that the tree needs to perform is to ensure that only leaf nodes can be edited. As you saw earlier in this chapter, it is possible to check that the node being edited is a leaf from within the cell editor (refer to "Controlling Editability in the Tree Cell Editor"), but it is simpler to make this check in the `JTree`

isPathEditable method. To override this method, you have to subclass JTree. The code is very simple:

```
JTree t = new JTree(m) {
  public boolean isPathEditable(TreePath path) {
    // Only allow editing of leaf nodes
    return isEditable() &&
      getModel().isLeaf(path.getLastPathComponent());
  }
};
```

This code is general enough that you can use it any time you want to ensure that only leaf nodes can be edited. Note that it uses the TreeModel isLeaf method to perform the test instead of the TreeNode isLeaf method, because the former calls the TreeNode getAllowsChildren method instead of isLeaf if the TreeModel has been set up to require this.

The Cell Editor

Constructing the cell editor is also a simple matter—all we need is an editor that displays a combo box containing the list of hardware items that can appear in the tree. The combo box itself can be created by calling the Tree-Model's getNamesAsCombo method. Therefore, creating and installing a suitable editor requires only four lines of code:

```
// Define a replacement editor
JComboBox combo = m.getNamesAsCombo();
DefaultCellEditor comboEditor =
                  new DefaultCellEditor(combo);
DefaultTreeCellEditor editor =
        new DefaultTreeCellEditor(t, r, comboEditor);
t.setCellEditor(editor);
```

The basic editor is created by wrapping the combo with a standard DefaultCellEditor, which is then wrapped in a DefaultTreeCellEdi-tor to make it behave suitably within a JTree. In this code extract, the variable r refers to the custom cell renderer for the tree that will be discussed next. Even though the editing process requires a certain amount of clever handling to arrange for the String from the combo box to be converted to the correct HardwareObject to be installed in the node, none of this complication is visible to the editor because it is all implemented in the Tree-Model. Therefore, it is perfectly acceptable to use a standard editor.

The Cell Renderer

Finally, let's look at the cell renderer. This needs to be customized because it has to display a specific icon for each node that has one and, conversely, it should display the standard icon if it comes across a node that does not have a special icon—that is, a node whose user object is not an instance of a class that implements the NodeWithIcon interface. In an earlier example, you saw how to change the icons that the tree uses for leaf nodes and for open and closed branch nodes by setting properties of the tree's DefaultTree-CellRenderer. However, these changes affect every icon in the tree. In this example, we need to be able to render a different icon for each leaf node. In order to do this, you need a custom renderer. Listing 10-6 showed how to create a renderer that uses a different color for the text of leaf and branch nodes. By expanding this example a little, it would be simple to have it use a different icon for each node as well - all that is necessary is to override the getLeafIcon method to return an icon that is suitable for the node being rendered. However, the approach taken in Listing 10-6 limits you to look-and-feel classes whose cell renderers are instances of DefaultTreeCellRen-derer or are subclasses of DefaultTreeCellRenderer that don't add any functionality to it that can't be lost. In many cases, of course, this is an acceptable constraint, but there is a technique that can be used to avoid being concerned about how the original renderer is implement. The code shown in Listing 10-8 shows how to create a renderer that will work in tandem with any installed renderer, as long as that renderer returns a JLabel from its getTreeCellRendererComponent method.

The new renderer works by obtaining a reference to the renderer that's installed in the tree when it is created, installing a reference to itself in its place and then calling the original renderer's getTreeCellRendererCom-ponent method at the start of its own. This allows the original renderer to create the component in its usual format, while allowing the new renderer to add its own customizations. If the component that the original renderer created is a JLabel and our node has an icon of its own, the only thing left to do is get the node's icon and install it in the original renderer's JLabel. There is no need to be concerned about how the label is organized, because this is a simple trade of one icon for another. You'll find the implementation details in Listing 10-8.

There is one small point in this renderer that is worth taking note of. The `getTreeCellRendererComponent` method replaces the icon in the `JLa-bel` created by the original renderer like this:

```
if (leaf && c instanceof JLabel) {
  JLabel l = (JLabel)c;
  Icon icon = getLeafIcon();
  if (icon != null) {
    l.setIcon(icon);
  }
}
```

In fact, we know that the Icon to be used is readily available to the `getTreeCellRendererComponent` method, because it is part of the node that is being rendered, assuming that the node's user object implements the `NodeWithIcon` interface (and if it doesn't, the original icon should be left in place). However, instead of extracting the icon directly, the task is delegated to the `getLeafIcon` method. The reason for this is actually to do with editing. As you know, in order to keep the correct icon displayed during an editing session, the `DefaultTreeCellEditor` constructor is given a reference to the tree's renderer. When the editor is activated, the renderer's `getLeaf-Icon` method is called to get the correct icon. For this reason, our renderer is obliged to supply a `getLeafIcon` method that gets the appropriate icon for whichever node is being edited and, since we had to implement this method, we chose to use it in the `getTreeCellRendererComponent` method as well. Here is how it is implemented:

```
public Icon getLeafIcon() {
  Object o = ((DefaultMutableTreeNode)lastValue).
               getUserObject();
  if (o instanceof NodeWithIcon) {
    return ((NodeWithIcon)o).getIcon();
  } else if (oldRenderer instanceof
         DefaultTreeCellRenderer) {
     return ((DefaultTreeCellRenderer)oldRenderer).
                    getLeafIcon();
  }
  return null;
}
```

There is a small subtlety in this code. The icon has to be obtained from the node being rendered (or edited in the case of an editing session), but the `DefaultTreeCellRenderer getLeafIcon` method does not receive a reference to this node. As with our custom editor implementation that was shown in "Controlling Editability in the Tree Cell Editor, " we solve this problem by storing a reference to the last node that was rendered in the `getTreeCellRendererComponent` method and then use it in `getLeafI-con`. This works both during normal rendering and when an editing session is in progress.

The rest of the code for this example just creates an instance of the tree with suitable data and displays it. Figure 10-15 shows what it looks like with one hardware item selected. If you run this example by typing the command:

```
Java JFCBook.Chapter10.CustomIcons
```

Figure 10-15 A tree node with a custom icon.

you can try out editing the leaf node by triple-clicking on it. You'll get a combo box with two selections in it. Notice how the text and the image both change when you change the selection. Notice also that the nonleaf nodes are not editable.

Summary

This chapter introduced the Swing tree control, one of two powerful data display components that you have at your disposal. After seeing the basic components of a tree, you learned how trees are created and how the user can manipulate them by expanding and collapsing nodes and making selections. You also saw how these actions are reflected in the tree itself and how they can be intercepted by applications and by custom components derived from the `JTree` class.

One of the more popular uses for a tree control is to display the content of a file system. A large portion of this chapter was devoted to creating a subclass of `JTree` that can show a file system in hierarchical form. The implementation of this control reveals much of the inner workings of the tree's data structures.

Finally, you saw several ways in which the tree's appearance can be customized to suit the needs of

particular applications and discovered how to allow the user to edit the contents of a tree, while still making sure that only meaningful changes can be made.
