MiscFile

Inherits From:	MiscGraphNode
Declared In:	misckit/MiscFile.h

Class Description

This class builds upon a MiscGraphNode so we can represent a file or hierarchy of files in the filesystem. You can create an instance of MiscFile using **initWithPath:** or **initWithName:inDirectory:**. Additionally, only one instance is allowed for a given pathname. Therefore if you already have a MiscFile instance representing "/LocalApps" and you ask for another, **initWithPath:** will free the receiver and return the first instance. Reference counting was added to keep track of duplicate instances so you don't have to worry about freeing the shared instances before you are done with all of them. As an example, you can do the following:

```
localApps = [ [MiscFile alloc] initWithPath: "/LocalApps"];
localApps2 = [ [MiscFile alloc] initWithPath: "/LocalApps"];
// Both localApps and localApps2 point to the same instance, but reference count = 2.
localApps = [localApps free];
```

```
// The instance is not really freed. reference count = 1.
localApps2 = [localApps2 free];
// The instance is freed since the reference count drops to 0.
```

This will be taken out for MiscKit 2.x.y, which will be built upon the NeXT Foundation Kit, which has its own reference counting scheme.

You also have considerable freedom in deciding what class your children (if you represent a directory) will be created from. First you can use **setDefaultClass:** *classId* to create all MiscFile's children as *classId*. You can set the class for individual subclasses of MiscFile using **setDefaultClass:***classId* **forClass:***miscFileClass.* This will use *classId* for creating children of a node that is a *miscFileClass* class. You can also choose your children's class based on it's filename extension. Therefore you can have *.app files instantiated from MiscAppWrapperFile, while *.bundle files being instantiated from MiscBundleFile, etc. To do this you have to register the extension with the class method **registerFileClass:forExtension:** You will also have to explicitly set that you would like to use the registered extensions via **setCreateChildrenUsingExtensions: YES**.   These methods allow great flexibility for creating a tree of nodes that have specific methods for specific kinds of files.

This looks quite complex, so an example is in order. One may want a tree of MiscFiles to represent a work directory containing NEXTSTEP projects. The work directory could be a MiscFile with sub directories of a class called called Project, the default class for Project could be set to ProjectFile to represent most files, with *.lproj represented by Localization classes, *.subproj by Project classes, *.tiff by TIFFfile classes that know how to work with images, *.[cmh] with CSourceFile classes etc. By careful registration of default classes with MiscFile, all of this can happen automatically. This is a very powerful method for creating an object that not only represents the directory tree, but knows how to do 'stuff' with each kind of file.

Here's some example code that will acheive this assuming that the classes above have been defined.

// We need to do this somewhere, -appDidInit   some -awakeFromNib or +initialize is good

```
- appDidInit:sender
    {
    ....
    // We want the top level (work directory) to have children of class Project
    [MiscFile setDefaultClass:[Project class] forClass:[MiscFile class]];

    // We want all project files to be of class ProjectFile unless they have a known extension
    [MiscFile setDefaultClass:[ProjectFile class] forClass:[Project class]];

    // *.subproj are SubProject class
    [MiscFile registerFileClass:[Project class] forExtension:"subproj"];

    // *.lproj are Localization class
    [MiscFile registerFileClass:[Localization class] forExtension:"subproj"];

    // *.tiff are TIFFfile class
    [MiscFile registerFileClass:[TIFFfile class] forExtension:"tiff"];
    ...
    }
```

Once this is done, doing something like

```
theWorkDirectory = [[MiscFile alloc] initWithPath:aWorkDir];
```

will create the appropriate classes for each file in the directory tree without further work whenever children are loaded.

Oh, and thanks very much to all the people who contributed ideas for this class and it's categories.... Bill

Bumgarner, Thomas Engel, Daniel Green, David Hunt, Frederic Stark, Alastair Thomson, Georg Tuparev, Don Yacktman, and Stephan Wacker. If I've missed anyone, I apologize. Enjoy the classes and if you find a bug or have a suggestion, please let me know.

Be sure to also read the documentation for the various MiscFile categories (Creation, Info, Modification, Searching, and UNIX).

Instance Variables

```
char *path;
struct stat *st;
short errorCode;
short refCount;
struct __mfFlags
{
        unsigned int childrenLoaded:1;
        unsigned int useExtensions:1;
        unsigned int displayAsLeaf:1;
        unsigned int childrenInherit:1;
        unsigned int visited:1;
        unsigned int enableCaching:1;
        unsigned int lastStat:1;
}
```

path	The pathname that the MiscFile represents.
st	Stat information for the file.
errorCode	A global error code for file information errors (something like errno).
refCount	Keep track of our instances.
mfFlags.childrenLoaded	Have our children been loaded yet.
mfFlags.useExtensions	Whether the children's filename extensions should determine their class.
mfFlags.displayAsLeaf	Give non-leaf nodes the choice to look like they are leaf nodes.
mfFlags.childrenInherit	Allows children to inherit attributes from their parents upon creation.
mfFlags.visited	Stops circular references when searching through MiscFile hierarchies.
mfFlags.enableCaching	Should stat the file every time or just use information we already have?
mfFlags.lastStat	Which function, lstat or stat represents the information in the ivar st?

Method Types

Setting a child's class	+ lookupClassForFilename: + registerFileClass:forExtension: + setDefaultFileClass: + setDefaultFileClass:forClass:
Creating files that really exist	+ isPathValid:

Class delegate	+ classDelegate + setClassDelegate
Initialization	- initWithName:inDirectory: - initWithPath: ± free
Querying	- fullPath - filename - extension
Attributes	- setLeaf: - displayAsLeaf - setDisplayAsLeaf:
Children inheriting attributes	-childrenInherit - setChildrenInherit - passOnTraits
Dealing with children	- children ± updateChildren ± loadChildrenFromFilesystem - childrenLoaded - _childrensClassGivenFilename: - createChildrenUsingExtensions - setCreateChildrenUsingExtensions:

Class Methods

classDelegate
+ **classDelegate**

Returns the class delegate.   I needed something like this for the searching category and thought that there could be other uses for a single delegate for all the MiscFiles.

See also: + **setClassDelegate:**

isPathValid:
+ (BOOL)**isPathValid: (const char *)***fullPath*

Stats (stat(2)) the given *fullPath* to make sure that it does indeed exist in the filesystem. This is used because you are not allowed to create any instances for which *fullPath* is not a valid system filename. Returns **YES** if the *fullPath* does exist, else **NO**. On the other hand if you wanted a subclass (or this class via a category) to circumvent this behavior, you could just override this method to return YES no matter what pathname is given to it.

lookupClassForFilename:
+ **lookupClassForFilename:**(char *)*filenameOrPath*

Returns the class that is associated with the extension of the filename or full pathname. This would have had to been set earlier with **+registerFileClass:forExtension:**. If the parameter's extension is not registered, **nil** is returned.

See also: + **registerFileClass:forExtension:**

registerFileClass:forExtension:
+ **registerFileClass:** *classId*
forExtension:(const char *)*extension*

Allows you to specify the class a child is instantiated from depending upon it's filename extension. You can then use **lookupClassForFilename:** to return the class that matches the given extension. If you wish to use extensions to determine a new child's class for a given instance, you must also call **setCreateChildrenUsingExtensions: YES,** because the default is **NO**.

See also: + **lookupClassForFilename:**

setClassDelegate:
+ **setClassDelegate:** *aDelegate*

Sets the delegate for all MiscFile classes. This is really here for future expansion. If anyone makes up any categories that need to consult some object (like my file searching categories) or send notification messages different from the announcer in MiscGraphNode, this may just do the trick.

See also: + **classDelegate**

setDefaultFileClass:
+ **setDefaultFileClass:** *classId*

Another way besides registering extensions to create children of different classes, is to set a default class that the children will be instantiated from. Therefore the parent can be a MiscFile and all the children can be instantiated from MiscBehavingFiles. If you wish to use the same class as the parent, which is the default, call this method and pass it **nil**.

See also: + **setDefaultFileClass:forClass:**, + **registerFileClass:forExtension:**

setDefaultFileClass:forClass:

+ **setDefaultFileClass:** *classId*
  forClass: *miscFileClass*

If you want different subclasses of MiscFile to use different default classes for their children, use this method. The *classId* is stored in a global hash table with *miscFileClass* as the key. If there is no entry in the table for a particular class, then the class used by **setDefaultFileClass:** is used if it has been set, otherwise the objects own class is used.

See also: + **setDefaultFileClass:**, + **registerFileClass:forExtension:**

Instance Methods

children

- **children**

Overridden from MiscGraphNode to load the node's children from the filesystem when asked. Returns the list of children. Children are loaded only upon demand. When children are newly loaded, an announcement of either **didAddChild**: or **didAddChildren**: will be sent to all the node's users and listeners via MiscGraphNode's announcer object. Also when the filesystem is updated by using **updateChildren**, announcements will be made if to reflect any changes (new files or less files). See MiscGraphNode for the types of announcements.

See also: **± loadChildrenFromFilesystem**

childrenInherit

- (BOOL)**childrenInherit**

Returns whether newly created children will inherit some of their parent's attributes.

See also: ± setChildrenInherit:, ± passOnTraits:

childrenLoaded

- (BOOL)**childrenLoaded**

Returns **YES** if the node's children have already been loaded. This method is needed because the children are loaded lazily.

See also: - **children**

_childrensClassGivenFilename:

- **_childrensClassGivenFilename:**(const char *)*filenameOrPath*

An internally used method. Returns the class that the node's child, with the given filename or pathname, should be instantiated from. It checks if we should use it's extension to determine it's class. If not, or an extension match is not found, then the default class is returned if it is not **nil**. Otherwise the class of the parent is returned.

See also: +setDefaultFileClass:, +registerFileClass:forExtension:

createChildrenUsingExtensions

- (BOOL)**createChildrenUsingExtensions**

Returns YES if this instance should attempt to use the child's filename extension when determining what class the child should be instantiated from. The default is NO.

See also: +registerFileClass:forExtension:, + lookupClassForFilename:

displayAsLeaf

- (BOOL)**displayAsLeaf**

This method allows non-leaf nodes to show up in a browser as leaf nodes. This of course if useful for app, bundles and nib file wrappers. This method returns YES if the node is a leaf node. If the node is a non-leaf, then it will return whether it would like to look like a leaf. The default is NO.

See also: ± setDisplayAsLeaf:

extension

- (const char *)**extension**

Returns the extension, which is stripped off the instance variable *path*. NULL is returned if the pathname has no extension.

See also: ± fullPath, ± filename

free

- **free**

If decrementing the instance's reference count does not bring it to zero, then nothing is freed and nil is returned. Otherwise all resources are freed and nil is returned.

filename

- (const char *)**filename**

Returns the filename that the object represents. The filename is derived from the instance variable *path*.

See also: ± extension, ± fullPath

fullPath

- (const char *)**fullPath**

Returns the full pathname of the file that the MiscFile represents.

See also: ± extension, ± filename

initWithName:inDirectory:

- **initWithName:**(const char *)*filename*
  inDirectory:(const char *)*directory*

Just concatenates directory and filename to create the full pathname and calls the designated initializer **initWithPath:**.

See also: ± initWithPath

initWithPath:

- **initWithPath:**(const char *)*fullPath*

The designated initializer. For each pathname in the filesystem there can only exist one instance of this class. Therefore if an instance of "/LocalApps" already exists and you ask for another, you will get a pointer to the first instead of two separate instances. Also if *fullPath* does not exist, the instance is freed and **nil** is returned. Otherwise **self** is returned. If you subclass MiscFile and **initWithPath:**, make sure that [super initWithPath:] returns a valid value (not nil) before continuing with your initialization.

See also: ± initWithName:inDirectory:

loadChildrenFromFilesystem

- loadChildrenFromFilesystem

Loads and instantiates all the node's children. This is here just in case you would like to load the children differently or more efficiently than I did. If you do override this method make sure that you take into account which class the newly read-in children should be instantiated from. Returns **self**.

See also: ± children

passOnTraits:

- passOnTraits: *parent*

If **childrenInherit** returns **YES**, then some attributes of the parent will be passed onto the children when they are created. Currently the only attribute that is passed on is createChildrenUsingExtensions, so that the children will continue to use filename extensions to determine their children's class if their parents did. If you add more traits (in a subclass) that would be useful to pass onto your children, this would be the method to override and pass them on. Returns **self**.

See also: ± childrenInherit, ± setChildrenInherit:

setChildrenInherit:

- setChildrenInherit: (BOOL)*inherit*

Sets whether children will inherit attributes (via **passOnTraits**) from their parent when they are created. The default is YES. Returns **self**.

See also: ± childrenInherit, ± passOnTraits:

setCreateChildrenUsingExtensions:

- **setCreateChildrenUsingExtensions:**(BOOL)*useExts*

Set to YES if you would like the class of any new children to be determined by the child's filename extension. It is NO by default.

See also: ± createChildrenUsingExtensions

setDisplayAsLeaf:

- **setDisplayAsLeaf:**(BOOL)*asLeaf*

Set to YES if you would like a non-leaf node to appear to be a leaf node. The default is NO. Returns **self**.

See also: ± displayAsLeaf

setLeaf:

- **setLeaf:**(BOOL)*isALeaf*

Overridden from MiscGraphNode since it doesn't make sense to allow an instance of MiscFile to alternate between a leaf and non-leaf. It is either a directory or it isn't. Returns **self** (and does nothing else).

See also: ± isLeaf (MiscGraphNode)