Interface Builder Kit Framework Reference

Tools & Languages: IDEs

2007-04-18

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction **Introduction 5**

Introduction 5

Part I **Classes 7**

Chapter 1 **IBDocument Class Reference 9**

Overview 9
Tasks 10
Class Methods 11
Instance Methods 11

Chapter 2 **IBInspector Class Reference 19**

Overview 19
Tasks 20
Class Methods 21
Instance Methods 21

Chapter 3 **IBPlugin Class Reference 25**

Overview 25
Tasks 25
Class Methods 26
Instance Methods 27

Chapter 4 **NSObject Interface Builder Kit Additions Reference 31**

Overview 31
Tasks 31
Instance Methods 33
Constants 40

Chapter 5 **NSView Interface Builder Kit Additions Reference 43**

Overview 43
Tasks 43
Instance Methods 44

Part II Data Types 49

Chapter 6 Interface Builder Kit Data Types Reference 51

Data Types 51

Part III Constants 53

Chapter 7 Interface Builder Kit Constants Reference 55

Constants 55

Document Revision History 57

Introduction

Framework	/<Xcode>/Library/Frameworks/InterfaceBuilderKit.framework
Header file directories	/<Xcode>/Library/Frameworks/InterfaceBuilderKit.framework/Headers
Companion guide	Interface Builder Plug-In Programming Guide
Declared in	IBDefines.h

Introduction

The Interface Builder Kit is a framework containing the classes you use to implement custom plug-ins for Interface Builder. A plug-in injects one or more custom objects into Interface Builder's library window. From the library window, users can access your custom objects and drag them into their nib files just as they would the standard system controls. You can also use this framework to implement inspectors for manipulating your objects at runtime.

The Interface Builder Kit is part of Xcode and as such is installed with the other developer tools in your Xcode installation directory and not in the `/System/Library/Frameworks` directory. The default installation for Xcode is the `/Developer` directory.

Introduction

Classes

PART I

Classes

IBDocument Class Reference

Inherits from	NSDocument : NSObject
Conforms to	`NSObject` (`NSObject`)
Framework	/Developer/Library/Frameworks/InterfaceBuilderKit.framework
Declared in	InterfaceBuilderKit/IBDocument.h
Companion guide	Interface Builder Plug-In Programming Guide

Overview

An `IBDocument` object provides the in-memory representation of a nib file and is one of the core classes in Interface Builder. It manages all of the objects in a nib file, observes them for undo support, and facilitates the general exchange of information with other objects. You use this object as is and do not subclass it.

A document object owns all of the objects in a nib file, including the windows, views, controls, controller objects, and custom objects in that nib file. You can use the methods of `IBDocument` to access these objects, add and remove objects, or change the inter-object relationships.

Most Interface Builder plug-ins should never need to access the current `IBDocument` object. The only times you might use this object are when you need to make additional changes to the object hierarchy beyond those provided automatically by Interface Builder. For example, you could use the document object to associate additional objects with a view as part of its design-time configuration.

Top-Level Objects

Each document contains a distinct set of top-level objects (also known as the "root objects"). These are the objects that appear at the top level of the document window and often consist of windows, menus, and custom controller objects. Placeholder objects such as File's Owner, First Responder, and Application are also top-level objects. (Placeholder objects are programmatic entities provided by Interface Builder.)

The user can add other top-level objects to a nib file by dragging items from the library window to the document window. You can add top-level objects programmatically using the methods of this class and specifying a parent object of `nil`.

First Class Objects

Interface Builder documents distinguish between objects that can be edited and selected and those that cannot. If an object can be edited and selected, it is considered to be a "first-class" object and is exposed to the user through Interface Builder's outline view. All other objects are "second-class", are controlled by an owning first-class object, and cannot be manipulated directly by the user in Interface Builder. Your plug-in can modify the second-class objects for which it is responsible. You should not attempt to modify second-class objects managed by other plug-ins, however.

All objects in a document, regardless of whether they are first class or second class, are stored in the same nib file. The distinction between first and second-class objects only affects access to the object in the Interface Builder windows and user interface.

Scroll views provide a perfect example of how first and second-class objects may be related in a nib file. A scroll view exposes its document view and scroller objects as first-class objects but does not expose its clip view. The clip view is considered to be an integral part of the scroll view and is therefore a second-class object to be hidden from the user. When the nib file is loaded, however, the clip view is instantiated along with the scroll view, document view, and scrollers.

Tasks

Getting the Document Object

`+ documentForObject:` (page 11)
Returns the document object that currently owns the specified object.

Getting the Objects in a Document

`- objects` (page 15)
Returns all of the first-class objects in the receiver.

`- topLevelObjects` (page 17)
Returns the top-level objects in the receiver.

`- childrenOfObject:` (page 12)
Returns an array containing the first-class children of the specified object.

`- parentOfObject:` (page 15)
Returns the first-class parent of the specified object.

`- removeObject:` (page 16)
Removes the specified object and its children from the document.

`- addObject:toParent:` (page 11)
Adds the specified object to the document as a first-class object.

`- moveObject:toParent:` (page 14)
Changes the parent of the specified object.

Getting the Object's Attributes

- `documentImageNamed:` (page 14)
  Returns the image resource with the specified name.
- `nameForDocumentImage:` (page 15)
  Returns the name of a specified image resource.
- `setMetadata:forKey:ofObject:` (page 16)
  Associates a custom value with the specified key of an object.
- `metadataForKey:ofObject:` (page 14)
  Returns the value associated with the specified key and object.

Creating Connections

- `connectOutlet:ofSourceObject:toDestinationObject:` (page 13)
  Creates an outlet connection between two objects.
- `connectAction:ofSourceObject:toDestinationObject:` (page 12)
  Creates an action connection between two objects.
- `connectBinding:ofSourceObject:toDestinationObject:keyPath:options:` (page 13)
  Creates a binding between two objects.

Class Methods

documentForObject:

Returns the document object that currently owns the specified object.

```
+ (id)documentForObject:(id)object
```

Parameters

object
The object whose document you want.

Return Value
The document object that owns the specified object, or `nil` if the object has no owner. Objects that are in transition (such as on the pasteboard) are not owned by a document.

Instance Methods

addObject:toParent:

Adds the specified object to the document as a first-class object.

```
- (void)addObject:(id)object toParent:(id)parent
```

Parameters

object

The object to add to the document.

parent

The parent of *object*, or `nil` if *object* should be added to the document as a top-level object. The parent object must be a first-class object of the document.

Discussion

This method adds the object to the document's internal data structures as a child of *parent*. This method does not create any object-specific associations between *object* and *parent*. For example, if both *parent* and *object* are views, this method does not configure *parent* as the superview of *object*. You must create any supplemental connections yourself.

See Also

- removeObject: (page 16)

childrenOfObject:

Returns an array containing the first-class children of the specified object.

```
- (NSArray *)childrenOfObject:(id)object
```

Parameters

object

The parent object whose children you want.

Return Value

An array of objects, each of which is a first-class object of *object*. If *object* has no children, this method returns an empty array object.

See Also

- parentOfObject: (page 15)

connectAction:ofSourceObject:toDestinationObject:

Creates an action connection between two objects.

```
- (void)connectAction:(NSString
    *)actionofSourceObject:(id)sourcetoDestinationObject:(id)destination
```

Parameters

action

The name of the action message. This string must correspond to a selector in the destination object and the selector name must end with a trailing colon character.

source

The object that receives the specified action message, otherwise known as the target of the action. This object should contain a selector whose signature matches the string in *action*.

destination

The object that triggers the action and sends the action message. This is typically a control, such as a button.

Discussion
Although the behavior of the *source* and *destination* parameters might seem reversed, they are not. The parameter names correspond to the order in which the action connection is established inside the plug-in, rather than the order in which action messages flow. In Interface Builder, action connections can be created starting at either end of the connection. For plug-ins, it is often more convenient to start at the object containing the action method and connect that method to the object that sends the action message.

connectBinding:ofSourceObject:toDestinationObject:keyPath:options:

Creates a binding between two objects.

```
- (void)connectBinding:(NSString
    *)bindingNameofSourceObject:(id)sourcetoDestinationObject:(id)destinationkeyPath:(NSString
    *)keyPathoptions:(NSDictionary *)options
```

Parameters

bindingName
The key path that identifies a property of the source object. The specified property must be bindable.

source
The source object that owns the property identified by the *bindingName* parameter.

destination
The object that provides data for the specified binding.

keyPath
The key path in the destination object that points to the data being bound.

options
A dictionary containing options for the binding, such as placeholder objects or an `NSValueTransformer` identifier. This value is optional—pass `nil` to specify no options. For information about the keys you can place in the dictionary, see the binding options constants in *NSKeyValueBindingCreation Protocol Reference*.

Discussion
For information about how to make your own custom objects bindable, see *Cocoa Bindings Programming Topics*.

connectOutlet:ofSourceObject:toDestinationObject:

Creates an outlet connection between two objects.

```
- (void)connectOutlet:(NSString
    *)outletofSourceObject:(id)sourcetoDestinationObject:(id)destination
```

Parameters

outlet
The name of the outlet.

source
The source object that contains the specified outlet. The specified outlet must exist and have the specified name.

destination
The object to assign to the specified outlet.

documentImageNamed:

Returns the image resource with the specified name.

```
- (NSImage *)documentImageNamed:(NSString *)name
```

Parameters

name

The name of the desired image. The image must be a resource in the receiving document. This parameter must not be `nil`.

Return Value

The image resource with the specified name, or a default "missing image" resource if an image with the specified name could not be found. If *name* contains an empty string, this method returns `nil`.

See Also

- nameForDocumentImage: (page 15)

metadataForKey:ofObject:

Returns the value associated with the specified key and object.

```
- (id)metadataForKey:(NSString *)keyofObject:(id)object
```

Parameters

key

The key used to identify the metadata value.

object

The first-class object containing the metadata value.

Return Value

The current value associated with the specified key.

See Also

- setMetadata:forKey:ofObject: (page 16)

moveObject:toParent:

Changes the parent of the specified object.

```
- (void)moveObject:(id)objecttoParent:(id)parent
```

Parameters

object

The object whose parent you want to change. This object must be a first-class object.

parent

The new parent for *object*. This object must be a first-class object. Specify `nil` to make *object* a top-level object of the document.

Discussion
This method changes the parentage of the object in the document's internal data structures. This method does not make assumptions about the type of *object* and therefore does change any other associations between it and its new or former parent. For example, if *object* is a view, this method does not change the superview of *object* to *parent*. You must make such a change yourself.

nameForDocumentImage:

Returns the name of a specified image resource.

```
- (NSString *)nameForDocumentImage:(NSImage *)image
```

Parameters
image
The image resource whose name you want to obtain. This parameter must not be `nil`.

Return Value
The name of the specified image, or `nil` if the image is unnamed or is not a resource in the receiving document.

Discussion
This method returns the name of an image resource obtained through Interface Builder's synchronization mechanism. The name is suitable for exposing to the user in an `IBInspector` subclass.

See Also
`- documentImageNamed:` (page 14)

objects

Returns all of the first-class objects in the receiver.

```
- (NSArray *)objects
```

Return Value
An array containing all of the nib file's first-class objects.

Discussion
For information about what comprises a first-class object, see "First Class Objects" (page 10).

See Also
`- topLevelObjects` (page 17)

parentOfObject:

Returns the first-class parent of the specified object.

```
- (id)parentOfObject:(id)object
```

Parameters
object
The object whose parent you want. This object must be a first-class object of the document.

Return Value
The closest parent to *object* that is also a first-class object, or `nil` if *object* is a root object.

See Also
`- childrenOfObject:` (page 12)

removeObject:

Removes the specified object and its children from the document.

```
- (void)removeObject:(id)object
```

Parameters

object
The object to remove.

Discussion
During the removal process, this method makes no assumptions about the type of each removed object. As a result, this method removes the object only from the document's own internal data structures. This includes breaking any outlet, action, or binding connections between *object* and any other objects in the document. It does not include breaking relationships that are part of the object's own behavior. For example, this method does not disassociate a view from its superview. You must break any object-specific relationships yourself either before or after removing the object from the document.

See Also
`- addObject:toParent:` (page 11)

setMetadata:forKey:ofObject:

Associates a custom value with the specified key of an object.

```
- (void)setMetadata:(id)value forKey:(NSString *)key ofObject:(id)object
```

Parameters

value
The value you want to set, or `nil` if you want to clear the current value. The value should be of a type supported by property lists, such as `NSString`, `NSNumber`, `NSArray`, `NSDictionary`, `NSData`, and so on. For more information, see *Property List Programming Guide*.

key
The key used to identify the value.

object
The first-class object on which to set the value.

Discussion
You can associate custom metadata values with the first-class objects of a document. The values you add are saved with the object in the nib file and persist across undo and pasteboard operations. Setting a metadata value on an object is not an undoable action itself, however.

If a metadata entry with the same key already exists in *object*, this method replaces the old value with the value in the *property* parameter.

See Also
- `metadataForKey:ofObject:` (page 14)

topLevelObjects

Returns the top-level objects in the receiver.

```
- (NSArray *)topLevelObjects
```

Return Value
An array of the nib file's top-level objects.

Discussion
This method returns all of the top-level objects in a document, including user-created objects (such as windows and menus) that have no parent object of their own, and placeholder objects such as File's Owner, First Responder, and Application. For more information, see "Top-Level Objects" (page 9).

See Also
- `objects` (page 15)

IBInspector Class Reference

Inherits from	NSObject
Conforms to	`NSObject` (`NSObject`)
Framework	/System/Library/Frameworks/InterfaceBuilderKit.framework
Availability	Available in Mac OS X v10.0 through Mac OS X v10.4.
Declared in	InterfaceBuilderKit/IBInspector.h
Companion guide	Interface Builder Plug-In Programming Guide

Overview

The `IBInspector` class manages the display and synchronization of custom attributes for a particular class in the inspector window. If you are implementing a custom view, control, or object whose class has user-editable attributes, you should provide a custom subclass of `IBInspector`.

Subclassing Notes

An `IBInspector` object is a controller that manages a custom user interface you define. The main job of your inspector object is to synchronize the changes made in its user interface with the data values in the underlying objects and vice versa. Each inspector object you create manages the interface for a single slice in the attributes pane of the inspector window. If your custom objects derive from multiple custom parent classes, you would typically implement a separate inspector object for each class.

To manage the custom user interface for an inspector, you can use bindings or outlets and actions. When using bindings, the synchronization is automatic. When using outlets and actions, the inspector object should push changes from inspector controls to the current objects being inspected. Conversely, when the selection itself changes, Interface Builder sends your inspector a `refresh` message, which your inspector object can use to update its interface.

For more information on creating a custom inspector object, see *Interface Builder Plug-In Programming Guide*.

Methods to Override

To implement an inspector, you typically override the following methods:

- `viewNibName` (page 23)
- `refresh` (page 22)

Overriding the `refresh` method is necessary if you are using actions and outlets to synchronize your inspector interface or if you need to perform some additional actions in your inspector whenever the selection changes. If you use Cocoa bindings exclusively to synchronize your inspector user interface with the current selection, you do not need to override the `refresh` method.

You may also want to override the `label` method to provide a custom label for your inspector slice. If you do not override this method, Interface Builder provides a reasonable default.

Tasks

Getting the Shared Inspector Object

`+ sharedInstance` (page 21)
Returns the shared instance of your custom inspector object.

Getting the Inspector Attributes

`+ supportsMultipleObjectInspection` (page 21)
Returns a Boolean value indicating whether the receiver supports the selection of multiple objects.

`- document` (page 21)
Returns the document object that contains the currently inspected objects.

`- label` (page 22)
Returns the user-readable string to display in your inspector slice.

`- view` (page 23)
Returns the content view of the receiver.

`- viewNibName` (page 23)
Returns the name of the nib file containing the receiver's content view.

Getting the Inspected Objects

`- inspectedObjects` (page 22)
Returns the currently selected objects that should be inspected.

`- inspectedObjectsController` (page 22)
Returns an array controller you can use to bind to the inspected objects.

Updating the Inspector View

`- refresh` (page 22)
Notifies the receiver that some aspect of the selection changed.

Class Methods

sharedInstance

Returns the shared instance of your custom inspector object.

```
+ (id)sharedInstance
```

Return Value
Your shared inspector object.

Discussion
You do not need to override this method. The first time this method is called, Interface Builder creates an instance of your inspector object and caches it for future access.

supportsMultipleObjectInspection

Returns a Boolean value indicating whether the receiver supports the selection of multiple objects.

```
+ (BOOL)supportsMultipleObjectInspection
```

Return Value
`YES` if the receiver supports multiple selections; otherwise, `NO`. The default implementation of this method returns `YES`.

Discussion
Multiple selection applies only when the selected objects share a common class. Whenever possible, you should design your inspector's user interface to display appropriate information when multiple objects are selected. If for some reason, your inspector cannot support multiple selection, you should override this method and return `NO`.

Instance Methods

document

Returns the document object that contains the currently inspected objects.

```
- (IBDocument *)document
```

Return Value
The active document.

Discussion
The current document maintains information that might be useful to your inspector, such as the objects currently in the nib file and the relationships between those objects.

You should not override this method.

inspectedObjects

Returns the currently selected objects that should be inspected.

```
- (NSArray *)inspectedObjects
```

Return Value
The currently selected objects, or an empty set if no objects are selected.

Discussion
This method returns always returns the current selection, regardless of whether that selection contains zero, one, or multiple objects.

See Also
- `inspectedObjectsController` (page 22)

inspectedObjectsController

Returns an array controller you can use to bind to the inspected objects.

```
- (NSArrayController *)inspectedObjectsController
```

Return Value
The array controller for the selected objects.

See Also
- `inspectedObjects` (page 22)

label

Returns the user-readable string to display in your inspector slice.

```
- (NSString *)label
```

Return Value
A string describing the receiver. By default, this method returns a formatted version of the receiver's class name.

Discussion
You can override this method to return a custom label for your inspector.

refresh

Notifies the receiver that some aspect of the selection changed.

```
- (void)refresh
```

Discussion
You should override this method in your inspector classes if you want to synchronize your inspector's content view manually with the values in the currently selected objects. In your implementation of this method, you should get the currently selected objects and use their current values to update the controls in the receiver's content view. You do not need to implement this method if you synchronize your inspector contents exclusively using Cocoa bindings.

If you override this method, you must call `super` at some point in your implementation.

view

Returns the content view of the receiver.

```
- (NSView *)view
```

Return Value
The content view containing the inspector's visible content. This view typically contains controls used to display the attributes of the currently inspected object.

Discussion
You do not need to override this method if your view is connected to the `inspectorView` outlet of your inspector class. (This outlet is exposed by the `IBInspector` class.) If your view is connected to that outlet, this method returns your view automatically. If you do not connect your view to that outlet, you must override this method and return your view object.

See Also
- `viewNibName` (page 23)

viewNibName

Returns the name of the nib file containing the receiver's content view.

```
- (NSString *)viewNibName
```

Return Value
A string identifying the receiver's nib file. This string should not include the `.nib` extension or any pathname information. Interface Builder looks for the specified nib file in the `Resources` directory of the plug-in bundle.

Discussion
You should override this method to return the name of the nib file containing your inspector's content view. If your inspector uses a single content view, you should set the File's Owner of the nib file to your custom `IBInspector` subclass and connect the `inspectorView` outlet of that class to your view. If your inspector supports multiple views, you should use a single master view and swap out its contents as needed when the inspector is refreshed.

If you prefer not to use a nib file to load your view, you can return `nil` from this method. If you do so, however, you must override the `view` method to return your view.

See Also
- `refresh` (page 22)
- `view` (page 23)

IBPlugin Class Reference

Inherits from	NSObject
Conforms to	`NSObject` (`NSObject`)
Framework	/System/Library/Frameworks/InterfaceBuilderKit.framework
Declared in	InterfaceBuilderKit/IBPlugin.h
Companion guide	Interface Builder Plug-In Programming Guide

Overview

The `IBPlugin` class provides the basic functionality required by all Interface Builder plug-ins. The methods of this class provide hooks for you to configure your plug-in when it is loaded by Interface Builder.

Subclassing Notes

Interface Builder uses this class as the entry point to your plug-in bundle. You must implement a custom subclass that provides basic information about your plug-in to Interface Builder.

Methods to Override

There is only one method you are required to override in this class:

- `libraryNibNames` (page 28)

You may override other methods of this class to support the initialization and customization of your plug-in object, but doing so is optional.

Tasks

Getting the Shared Plug-in Object

`+ sharedInstance` (page 26)
Returns the shared plugin object.

Loading and Unloading Plug-in Resources

- `didLoad` (page 27)
  Notifies the receiver that it has been loaded into the Interface Builder environment.
- `willUnload` (page 29)
  Notifies your plug-in that it has been removed from the Interface Builder environment.

Getting the Plug-in's Custom Objects

- `libraryNibNames` (page 28)
  Returns an array of strings containing the names of your plug-in's custom nib files.

Configuring Your Plug-in

- `label` (page 27)
  Returns the user-readable name displayed for your plug-in object in the Interface Builder preferences window.
- `preferencesView` (page 28)
  Returns the custom view used to display your plug-in's preferences.
- `requiredFrameworks` (page 29)
  Returns the framework bundles required by your plug-in code to operate.

Pasteboard Notifications

- `pasteboardObjectsForDraggedLibraryView:` (page 28)
  Notifies the receiver that one of its library items is about to be added to a document.
- `document:didAddDraggedObjects:fromDraggedLibraryView:` (page 27)
  Notifies the receiver that one or more objects were added to the specified document from the library.

Class Methods

sharedInstance

Returns the shared plugin object.

```
+ (id)sharedInstance
```

Return Value
The shared `IBPlugin` object.

Discussion
You do not need to override this method. You may call it at any time from your plug-in code to retrieve your own plug-in object.

Instance Methods

didLoad

Notifies the receiver that it has been loaded into the Interface Builder environment.

```
- (void)didLoad
```

Discussion
You can override this method to initialize any variables or resources that your plug-in requires whenever it is loaded by Interface Builder. If you do override this method, you must call `super` at some point in your implementation.

If you implement this method, you should not rely on Interface Builder calling the `willUnload` method to undo your plug-in object's initialization. If your `didLoad` method acquires any resources that must be freed later, you should release those resources in your plug-in object's `dealloc` or `finalize` method instead.

See Also
- `willUnload` (page 29)

document:didAddDraggedObjects:fromDraggedLibraryView:

Notifies the receiver that one or more objects were added to the specified document from the library.

```
- (void)document:(IBDocument *) documentdidAddDraggedObjects:(NSArray *)
    rootsfromDraggedLibraryView:(NSView *) view
```

Parameters

document
The document that received the specified objects.

roots
The root objects that were added to the document. This array does not contain any child objects attached to the root objects.

view
The library view (owned by the receiver) from which the objects were added.

Discussion
You can use this method to create additional associations, connections, or bindings among a group of objects after they are added to a document. For example, you might use this method to create connections that are not present in the library view version of the objects.

label

Returns the user-readable name displayed for your plug-in object in the Interface Builder preferences window.

```
- (NSString *)label
```

Return Value
A string containing the user-readable name of your plug-in.

Discussion
If you do not provide a name for your plug-in, the default implementation returns a formatted version of the receiver's class name by default.

libraryNibNames

Returns an array of strings containing the names of your plug-in's custom nib files.

```
- (NSArray *)libraryNibNames
```

Return Value
An array of `NSString` objects, each of which corresponds to the name of a nib file in your plug-in's `Resources` directory. The nib file name does not require any leading path information or the `.nib` filename extension.

Discussion
The nib files you return should contain one or more library-object template views. These views act as containers for your own custom objects. Each library-object template view contains one or more objects to add the library window. For information on how to configure these views in your nib file, see *Interface Builder Plug-In Programming Guide*.

You must override this method in your plug-in subclass.

pasteboardObjectsForDraggedLibraryView:

Notifies the receiver that one of its library items is about to be added to a document.

```
- (NSArray *)pasteboardObjectsForDraggedLibraryView:(NSView *) view
```

Parameters

view

The view that was dragged from the library.

Return Value
The array of objects that should actually be added to the document. Only the root object of a given object hierarchy need be returned; you do not need to include any associated child objects of that root object in the array.

Discussion
You can use this method to replace a library view with the actual objects that the view represents. For example, you might use an image view in the library to provide a better visual representation of the underlying objects. In such a case, though, you would not want the image view to be added to a document. Instead, you would implement this method and use it to exchange the image view for the actual objects.

If you do not implement this method, the default version returns an array containing the object in the *view* parameter.

preferencesView

Returns the custom view used to display your plug-in's preferences.

```
- (NSView *)preferencesView
```

Return Value
The plug-ins custom preferences view.

Discussion
If your plug-in supports configurable preferences, you can override this method to return the view used to display those preferences. When your plug-in is selected in the Interface Builder preferences window, your custom view replaces the list of plug-ins and frameworks normally displayed for plug-ins.

requiredFrameworks

Returns the framework bundles required by your plug-in code to operate.

```
- (NSArray *)requiredFrameworks
```

Return Value
An array of `NSBundle` objects, each of which is initialized to a framework directory.

Discussion
Interface Builder uses the information returned from this method to load your plug-in's required frameworks during simulation. If your plug-in requires any custom frameworks, such as those containing your custom view code, you must override this method and return those frameworks.

Your implementation of this method should return only those frameworks containing your custom object code. You do not need to return any system frameworks your objects depend on.

willUnload

Notifies your plug-in that it has been removed from the Interface Builder environment.

```
- (void)willUnload
```

Discussion
You can override this method to handle any cleanup that might be required when your plug-in is removed from Interface Builder by the user. This method is called only when the user removes your plug-in from the list of plug-ins in the preferences window. It is not called when the Interface Builder application quits. If you do override this method, you must call `super` at some point in your implementation.

See Also
- `didLoad` (page 27)

NSObject Interface Builder Kit Additions Reference

Inherits from	NSObject
Conforms to	`NSObject` (`NSObject`)
Framework	/System/Library/Frameworks/InterfaceBuilderKit.framework
Declared in	InterfaceBuilderKit/IBObjectIntegration.h
Companion guide	Interface Builder Plug-In Programming Guide

Overview

The Interface Builder Kit framework extends the `NSObject` class by adding methods to discover design-time information about the object, such as its attributes and child objects. Additional methods provide Interface Builder with the classes used to inspect the object.

Subclassing Notes

You do not call the methods of this class directly. Instead, you override them to provide Interface Builder with information about your custom objects. Although all of the methods have a default implementation that you can use, many need to be overridden to provide relevant information about the receiver. Which methods you override is determined by the capabilities of your custom objects.

If your custom object is a subclass of `NSView`, you still probably need to override methods of this category to provide relevant object-level information. In addition, you may also need to override methods of the `NSView` category, which is described in *NSView Interface Builder Kit Additions Reference*.

Because these methods are relevant only in the context of Interface Builder, it is recommended that you implement these methods in a category on your class and include the category code only in your Interface Builder plug-in.

Tasks

Providing the Designable Attributes of an Object

`- ibPopulateKeyPaths:` (page 37)
Returns a set of key paths identifying the receiver's designable attributes.

Specifying the Inspectors of an Object

- `ibPopulateAttributeInspectorClasses:` (page 37)
  Returns the inspector classes used to edit the receiver's attributes.
- `ibPopulateSizeInspectorClasses:` (page 38)
  Returns the inspector classes used to edit the receiver's size.

Getting the Object's Name and Icon

- `ibDefaultLabel` (page 34)
  Returns the display name of the receiver.
- `ibDefaultImage` (page 34)
  Returns the icon of the receiver.

Identifying Child Objects in the View Hierarchy

- `ibObjectAtLocation:inWindowController:` (page 37)
  Returns the child object of the receiver that is located at the specified point in the window.

Removing Child Objects

- `ibCanRemoveChildren:` (page 33)
  Allows or disallows the removal of child objects.
- `ibRemoveChildren:` (page 39)
  Removes child objects from a parent.

Specifying Child Object Information

- `ibDefaultChildren` (page 34)
  Returns the array of objects that should be exposed as children of the receiver.
- `ibRectForChild:inWindowController:` (page 39)
  Returns the visible frame of the receiver's child objects.
- `ibIsChildInitiallySelectable:` (page 36)
  Returns a Boolean value indicating whether the specified child object should be selected before its parent.
- `ibIsChildViewUserMovable:` (page 36)
  Returns a Boolean value indicating whether the location of the specified child view can be changed by the user.
- `ibIsChildViewUserSizable:` (page 36)
  Returns a Boolean value indicating whether the size of the specified child view can be changed by the user.

Document-related Notifications

`- ibDidAddToDesignableDocument:` (page 35)
Notifies the receiver that it was added to the specified document.

`- ibDidRemoveFromDesignableDocument:` (page 35)
Notifies the receiver that it was removed from the specified document.

`- ibAwakeInDesignableDocument:` (page 33)
Notifies the receiver that it is about to be loaded into Interface Builder for the first time.

Instance Methods

ibAwakeInDesignableDocument:

Notifies the receiver that it is about to be loaded into Interface Builder for the first time.

```
- (void)ibAwakeInDesignableDocument:(IBDocument *)document
```

Parameters

document
The document into which the receiver was added.

Discussion
This method is called once when the object is first instantiated and added to a document in Interface Builder. You can use this method to perform any one-time configuration of the object.

See Also
`- ibDidAddToDesignableDocument:` (page 35)

ibCanRemoveChildren:

Allows or disallows the removal of child objects.

```
- (BOOL)ibCanRemoveChildren:(NSSet *)objects
```

Parameters

objects
The child objects that are candidates for removal.

Return Value
`YES` to allow removal of the specified children; otherwise, `NO` if one of the child objects should not be removed.

Discussion
Interface Builder objects are stored in the IBDocument tree, and also by their natural relationships. For example, a button in a box is a child of the box in the Interface Builder document tree, but it is also a subview of the box's content view. In this context, the remove action means to remove objects from their natural relationships, not from the document tree.

Interface Builder calls this method to determine whether it can remove a parent's child objects. Unless this method is used to specify otherwise, Interface Builder assumes it can remove an object. You should override this method if your object has a child object that should not be removed.

Your implementation of this method should never return `YES`. You should either return `NO` because you know that one of the objects cannot be removed, or you should call the superclass implementation of this method and return the result.

A typical implementation on `NSPopUpButtonCell` might look like this:

```
- (BOOL)ibCanRemoveChildren:(NSSet *)children {
    BOOL deny = [children containsObject:[self menu]];
    return deny ? NO : [super ibCanRemoveChildren:children];
}
```

See Also
`- ibRemoveChildren:` (page 39)

ibDefaultChildren

Returns the array of objects that should be exposed as children of the receiver.

```
- (NSArray *)ibDefaultChildren
```

Return Value
The array of objects you want to expose.

Discussion
Interface Builder assumes that only the receiver is a first-class object—that is, only the receiver can be edited and manipulated by the user. However, if the receiver's associated objects also have editable attributes, you can expose those objects using this method. To do so, you simply override this method and return an array containing the objects you want to expose. Objects exposed using this method are also considered first-class objects. (For a detailed description of what constitutes a first-class object, see *IBDocument Class Reference*.) This method is typically used to expose subviews in a view hierarchy but may be used to expose non-view objects.

In your implementation of this method, you should return only the immediate child objects of the receiver you want to expose. Interface Builder calls this method recursively on the objects you return to allow them to expose their own children. If you do not want to expose a child view, but do want to expose a grandchild view, you may return the grandchild view as if it belonged to the receiver. Essentially, a view should be exposed only by one ancestor.

ibDefaultImage

Returns the icon of the receiver.

```
- (NSImage *)ibDefaultImage
```

Return Value
An image object containing the receiver's icon.

ibDefaultLabel

Returns the display name of the receiver.

```
- (NSString *)ibDefaultLabel
```

Return Value
A string containing the receiver's user-presentable name.

Discussion
You can override this method to return a custom name for the receiver. The default implementation of this method uses the following rules to return a name based on the receiver's class name:

1. Leading capital letters are stripped from the class name, except for the last one.
2. Spaces are inserted whenever the case changes.
3. The leading letter of each secondary word is then changed to lower case.

For example, the class name `NSTabViewItem` becomes the string "Tab view item".

ibDidAddToDesignableDocument:

Notifies the receiver that it was added to the specified document.

```
- (void)ibDidAddToDesignableDocument:(IBDocument *)document
```

Parameters
document
The document to which the receiver was added.

Discussion
You can use this method to perform any per-document configuration of the receiver. If you only need to configure an object once when it is first instantiated, override the `ibAwakeInDesignableDocument:` method instead.

See Also
- `ibDidRemoveFromDesignableDocument:` (page 35)
- `ibAwakeInDesignableDocument:` (page 33)

ibDidRemoveFromDesignableDocument:

Notifies the receiver that it was removed from the specified document.

```
- (void)ibDidRemoveFromDesignableDocument:(IBDocument *)document
```

Parameters
document
The document from which the receiver was removed.

Discussion
You can use this method to undo any per-document configuration you performed in the `ibDidAddToDesignableDocument:` method.

See Also
- `ibDidAddToDesignableDocument:` (page 35)

ibIsChildInitiallySelectable:

Returns a Boolean value indicating whether the specified child object should be selected before its parent.

```
- (BOOL)ibIsChildInitiallySelectable:(id)child
```

Parameters

child

A child object of the receiver.

Return Value

`YES` if an initial click on the specified child object results in the selection of the child object instead of the parent; otherwise, `NO` if the parent should be selected before its child. This method returns `YES` by default.

ibIsChildViewUserMovable:

Returns a Boolean value indicating whether the location of the specified child view can be changed by the user.

```
- (BOOL)ibIsChildViewUserMovable:(NSView *)view
```

Parameters

view

A child view of the receiver.

Return Value

`YES` if the location of the child view is user modifiable ; otherwise, `NO`.

Discussion

In most cases, you should not need to override this method. The default implementation returns an appropriate value based on the type of the receiver and the corresponding child view. The only time you might ever need to override this method is if the receiver is a container view that limits the movement of its child views. For example, a scroll view pins its scrollers at particular locations within its frame and therefore returns `NO`.

ibIsChildViewUserSizable:

Returns a Boolean value indicating whether the size of the specified child view can be changed by the user.

```
- (BOOL)ibIsChildViewUserSizable:(NSView *)child
```

Parameters

child

A child view of the receiver.

Return Value

`YES` if the size of the child is user modifiable; otherwise, `NO`.

Discussion

In most cases, you should not need to override this method. The default implementation returns an appropriate value based on the type of the receiver and the corresponding child view. The only time you might ever need to override this method is if the receiver is a container view that limits the sizing behavior of its child views. For example, a scroll view does not allow its scrollers to be resized and therefore returns `NO`.

ibObjectAtLocation:inWindowController:

Returns the child object of the receiver that is located at the specified point in the window.

```
- (id)ibObjectAtLocation:(NSPoint)windowPoint inWindowController:(NSWindowController
    *)controller
```

Parameters

point

The point, in the window's coordinate space, to test for a child object.

controller

The window controller object that manages the window of the receiver.

Return Value

The object at the specified point, or `nil` if the point is not in the receiver or in any of its child objects.

Discussion

Interface Builder calls this method recursively to locate objects in a window. The default implementation of this method walks the view hierarchy to find objects and is sufficient for most custom objects. You should override this method only if your object has visible child objects that lie outside of the receiver's own frame.

ibPopulateAttributeInspectorClasses:

Returns the inspector classes used to edit the receiver's attributes.

```
- (void)ibPopulateAttributeInspectorClasses:(NSMutableArray *)classes
```

Parameters

classes

On input, a mutable array. On output, this array should contain the stack of inspector classes to use to inspect the receiver.

Discussion

You typically override this method for each custom object or view provided by your plug-in. Your implementation of this method should call `super` first to retrieve the inherited inspectors for your class. You should then add any custom inspector classes to the array in the order that you want the corresponding slices to appear in the inspector window. The last inspector added to the *classes* array appears at the top of the inspector window, with the other inspectors situated below it.

Each inspector class you add must be a subclass of the `IBInspector` class. The inspectors you add appear under the attributes inspector mode of the inspector window.

A fairly typical implementation of this method would look like the following:

```
- (void)ibPopulateAttributeInspectorClasses:(NSMutableArray *)classes
{
    [super ibPopulateAttributeInspectorClasses:classes];
    [classes addObject:[MyInspectorClass class]];
}
```

ibPopulateKeyPaths:

Returns a set of key paths identifying the receiver's designable attributes.

```
- (void)ibPopulateKeyPaths:(NSMutableDictionary *)keyPaths
```

Parameters

keyPaths

A mutable dictionary into which you can place the receiver's key-value observable key paths.

Discussion

This method lets you return a set of key paths that identify the attributes of your objects. Interface Builder monitors these key paths and uses them to provide automatic support for undo operations and for the application's "lift-and-stamp" feature, which copies attributes between different instances of your object. The objects whose attributes you are exposing must be key-value observing (KVO) compliant. For information on how to support key-value observing, see *Key-Value Observing Programming Guide*.

The `keyPaths` dictionary contains each of the keys described in "Key paths dictionary keys" (page 40). Each key is associated with an `NSMutableSet` object to which you can add your own key path strings. The following is a list of the types of key paths that should go into each set:

- Add the key paths to your object's scalar attribute values (such as strings, booleans, and integers) to the `IBAttributeKeyPaths` set. Scalar attribute values include values represented by classes such as `NSString`, `NSNumber`, `NSColor`, `NSDate`, `NSValue`, and so on.
- Add the key paths to your object's to-one relation attributes to the `IBToOneRelationshipKeyPaths` set. To-one relations represent connections to other objects that have their own attributes. For example, controls typically have a to-one relation to their cell object.
- Add the key paths to your object's to-many relation attributes to the `IBToManyRelationshipKeyPaths` set. An example of an attribute with a to-many relation is one that contains a collection object.
- For any scalar attributes that contain localizable strings, add the corresponding key paths to the `IBLocalizableStringKeyPaths` set. This group is typically a subset of your object's scalar attributes.
- For any scalar attributes that contain geometry-based information, add the corresponding key paths to the `IBLocalizableGeometryKeyPaths` set. Geometry based attributes might include integers or floating-point values that represent the size or position of the object or its contents. Geometry information can change during the localization process and Interface Builder uses this information to record all relevant changes.
- For attributes that are localizable but are not string or geometry-based values, add the corresponding key paths to the `IBAdditionalLocalizableKeyPaths` set. For example, you could use this set to store the key paths to attributes containing images whose content might require localization.

Before adding the key paths associated with the receiver's class, call the `super` implementation to add the attributes of any superclasses.

ibPopulateSizeInspectorClasses:

Returns the inspector classes used to edit the receiver's size.

```
- (void)ibPopulateSizeInspectorClasses:(NSMutableArray *)classes
```

Parameters

classes

On input, a mutable array. On output, this array should contain the stack of inspector classes to use to inspect the receiver.

Discussion
You typically override this method for each custom object or view provided by your plug-in. Your implementation of this method should call `super` first to retrieve the inherited inspectors for your class. You should then add any custom inspector classes to the array in the order that you want the corresponding slices to appear in the inspector window. The last inspector added to the *`classes`* array appears at the top of the inspector window, with the other inspectors situated below it.

Each inspector class you add must be a subclass of the `IBInspector` class. The inspectors you add appear in the size inspector pane of the inspector window.

A typical implementation of this method would look like the following:

```
- (void)ibPopulateSizeInspectorClasses:(NSMutableArray *)classes
{
    [super ibPopulateSizeInspectorClasses:classes];
    [classes addObject:[MyInspectorClass class]];
}
```

ibRectForChild:inWindowController:

Returns the visible frame of the receiver's child objects.

```
- (NSRect)ibRectForChild:(id)child inWindowController:(NSWindowController
    *)controller
```

Parameters

`child`

The child object of the receiver whose visible frame should be returned. This object may not necessarily be a view. For example, if the receiver is a control, this parameter could contain the control's cell.

`controller`

The window controller object that manages the window containing the receiver.

Return Value
The visible frame of the child object in the specified window, or `NSZeroRect` if the child does not belong to the receiver or is not in the window managed by *`controller`*. The returned rectangle must be in the window's coordinate space.

Discussion
You should override this method if the receiver embeds child objects that are not views but which occupy a portion of the receiver's frame or if the receiver embeds child views that lie outside of the receiver's own frame but are still visible. If all of the receiver's children are views and are enclosed completely within the receiver's frame, you do not need to override this method. Controls that embed cells might use this method to return the frame for each cell.

ibRemoveChildren:

Removes child objects from a parent.

```
- (void)ibRemoveChildren:(NSSet *)objects
```

Parameters

`objects`

The child objects that are candidates for removal.

Discussion
Interface Builder objects are stored in the IBDocument tree, and also by their natural relationships. For example, a button in a box is a child of the box in the Interface Builder document tree, but it is also a subview of the box's content view. In this context, the remove action means to remove objects from their natural relationships, not from the document tree.

Interface Builder calls this method to remove a parent's child objects. If your object has child objects, and you didn't override `ibCanRemoveChildren:` to block their removal, then you must override this method to remove them. In the override, the receiver is responsible for actually removing references to each of the passed in children.

Overrides of this method should check for the presence of their removable children in the passed in set. If the set contains the children, the children should be removed from the receiver's natural relationships. For example, if the receiver is an NSView-based object and the passed in set contains a subview, the receiver would typically remove that object from its subview list.

The passed in objects may be a heterogeneous group of children. For example, if a pop up button cell has a formatter and allows removal of its menu, the passed in objects could include both the formatter and the menu.

A typical implementation on `NSCell` might look like this:

```
- (void)ibRemoveChildren:(NSSet *)children {
    if ([self formatter] && [children containsObject:[self formatter]]) {
        [self setFormatter:nil];
    }
    [super ibRemoveChildren:children];
}
```

See Also
`- ibCanRemoveChildren:` (page 33)

Constants

Key paths dictionary keys

These variables identify the dictionary keys used to group and identify your code's available key paths.

```
extern NSString * const IBAttributeKeyPaths;
extern NSString * const IBToOneRelationshipKeyPaths;
extern NSString * const IBToManyRelationshipKeyPaths;
extern NSString * const IBLocalizableStringKeyPaths;
extern NSString * const IBLocalizableGeometryKeyPaths;
extern NSString * const IBAdditionalLocalizableKeyPaths;
```

Constants
`IBAttributeKeyPaths`
Identifies the scalar attributes of an object. Scalar attributes include `NSString`, `NSNumber`, `NSDate`, `NSValue`, and other objects that contain or represent scalar values.

`IBToOneRelationshipKeyPaths`
Identifies the attributes of an object that contain other related objects.

`IBToManyRelationshipKeyPaths`
: Identifies the attributes of an object that contain collections of other related objects.

`IBLocalizableStringKeyPaths`
: Identifies the scalar attributes of an object that contain localizable strings.

`IBLocalizableGeometryKeyPaths`
: Identifies the scalar attributes of an object that contain geometric information that can change during localization.

`IBAdditionalLocalizableKeyPaths`
: Identifies any localizable attributes that are not stored as strings or geometry-related scalar types.

Declared In
`InterfaceBuilderKit/IBObjectIntegration.h`

NSView Interface Builder Kit Additions Reference

Inherits from	NSResponder : NSObject
Conforms to	`NSObject` (`NSObject`)
Framework	/System/Library/Frameworks/InterfaceBuilderKit.framework
Declared in	InterfaceBuilderKit/IBViewIntegration.h
Companion guide	Interface Builder Plug-In Programming Guide

Overview

The Interface Builder Kit framework extends the `NSView` class by adding the following:

- Methods for determining the position of Aqua guides
- Methods for determining design-time size information

Interface Builder uses these methods to discover relevant information about the views provided by your plug-in. You typically do not call these methods directly. Instead, you override them to customize the design-time information for your views.

Because these methods are relevant only in the context of Interface Builder, it is recommended that you implement these methods in a category on your class and include the category code only in your Interface Builder plug-in.

Tasks

Specifying the Container View

`- ibDesignableContentView` (page 45)
Returns the container view for the children of the receiver.

Providing Suggested Sizing Information

`- ibPreferredDesignSize` (page 46)
Returns the recommended size for the receiver.

- `ibPreferredResizeDirection` (page 47)
  Returns the direction in which the receiver should grow when its size changes programmatically.

Providing Design-time Layout Information

- `ibLayoutInset` (page 45)
  Returns the inset values for the receiver.
- `ibMinimumSize` (page 46)
  Returns the minimum size of the receiver.
- `ibMaximumSize` (page 46)
  Returns the maximum size of the receiver.

Providing Baseline Information

- `ibBaselineCount` (page 45)
  Returns the number of baselines available for guide alignment in the receiver.
- `ibBaselineAtIndex:` (page 44)
  Returns the distance from the view's y origin to the specified baseline.

Instance Methods

ibBaselineAtIndex:

Returns the distance from the view's y origin to the specified baseline.

```
- (float)ibBaselineAtIndex:(int)index
```

Parameters

index
  The index of the desired baseline. This value is be between 0 and the number of baselines returned by the `ibBaselineCount` method.

Return Value
The distance from the view's y origin to the specified baseline, measured in the view's local coordinate system.

Discussion
Interface Builder calls this method to retrieve the baseline positions associated with the receiver. You should not need to call this method directly.

Baselines are used for the alignment of text in a view. Interface Builder uses this information to compute the locations of Aqua guides that appear during layout.

See Also
- `ibBaselineCount` (page 45)

ibBaselineCount

Returns the number of baselines available for guide alignment in the receiver.

```
- (int)ibBaselineCount
```

Return Value
The total number of baselines available with this view. The default implementation returns 0.

Discussion
Interface Builder calls this method to retrieve the number of baselines associated with the receiver. If your view defines any baselines other than its basic boundaries (plus any relevant inset), you should override this method and return the number of additional baselines.

ibDesignableContentView

Returns the container view for the children of the receiver.

```
- (NSView *)ibDesignableContentView
```

Return Value
The container view for the receiver's children, or `nil` if the view does not contain children.

Discussion
If you are implementing a container view, you should override this method and return the view used to contain any child views. For most views, you would simply return `self` from this method. If the receiver exposes child views but does not directly contain them, however, you should return the view that does contain the exposed child views. For example, a scroll view returns its clip view, which is the actual container for the document view.

ibLayoutInset

Returns the inset values for the receiver.

```
- (IBInset)ibLayoutInset
```

Return Value
A structure containing the inset values (measured in points) for each edge of the receiver. Positive inset values shrink the size of your view while negative values expand it.

Discussion
Inset values let you specify the amount by which to shrink your view's bounds rectangle in order to lay it out visually. You should provide custom inset boundaries if the visible part of your view does not occupy the entire view bounds. The inset values combined with the view bounds should result in a rectangle that just encloses the relevant portion of your view.

For example, imagine a control that contains a shadow effect. Suppose the control occupies the entire view except for a small border along the bottom and right side that contain a shadow. If the shadow extends 3 points to the bottom and right of the control, this method should return a `IBInset` structure with the values {0, 0, 3, 3}.

For more information about layout boundaries, see *Interface Builder Plug-In Programming Guide*.

ibMaximumSize

Returns the maximum size of the receiver.

```
- (NSSize)ibMaximumSize
```

Return Value
The maximum size of the receiver, specified using the view's own local (bounds) coordinate space.

Discussion
For views with no practical size limit, you can return an arbitrarily large value, such as 10000, for each dimension. This value affects only the design-time size of the view. You can still modify the size of the view at runtime using the view's `setFrame:` and `setSize:` methods.

The size values you return should use the same coordinate system as your view's bounds rectangle. Normally, a view's bounds rectangle is tied to its frame rectangle and coordinates are measured in points (1/72 of an inch). It is possible, however, to detach a view's bounds coordinates from its frame coordinates. You might do this so that your view can use a fixed coordinate system internally, regardless of the view's actual size. Interface Builder automatically converts the bounds coordinates you return to window coordinates before performing any layout.

ibMinimumSize

Returns the minimum size of the receiver.

```
- (NSSize)ibMinimumSize
```

Return Value
The minimum size of the receiver, specified using the view's own local (bounds) coordinate space.

Discussion
This value affects only the design-time minimum size of the view. You can still modify the size of the view at runtime using the view's `setFrame:` and `setFrameSize:` methods.

The size values you return should use the same coordinate system as your view's bounds rectangle. Normally, a view's bounds rectangle is tied to its frame rectangle and coordinates are measured in points (1/72 of an inch). It is possible, however, to detach a view's bounds coordinates from its frame coordinates. You might do this so that your view can use a fixed coordinate system internally, regardless of the view's actual size. Interface Builder automatically converts the bounds coordinates you return to window coordinates before performing any layout.

ibPreferredDesignSize

Returns the recommended size for the receiver.

```
- (NSSize)ibPreferredDesignSize
```

Return Value
The suggested size for this view. This method returns the view's current size by default.

ibPreferredResizeDirection

Returns the direction in which the receiver should grow when its size changes programmatically.

```
- (IBDirection)ibPreferredResizeDirection
```

Return Value
The suggested growth direction. For a list of possible values, see *Interface Builder Kit Constants Reference*.

Discussion
Programmatic changes to the size of a view might can come from Interface Builder or from the user changing values in the inspector. System controls return the suggested horizontal growth direction associated with their cells by default.

Data Types

Interface Builder Kit Data Types Reference

Data Types

IBInset

The `IBInset` structure stores the values by which to inset a rectangle.

```
typedef struct IBInsetTag {
    float left;
    float top;
    float right;
    float bottom;
} IBInset;
```

Discussion
This structure is used during layout to adjust a view's frame rectangle to compensate for adornments. For example, controls use these values to ensure that layout occurs along the edge of the actual control and not along its shadow. Positive values shrink the view's frame while negative values expand it.

Availability
Available in Mac OS X v10.5 and later.

Declared In
`IBDefines.h`

Constants

PART III

Constants

Interface Builder Kit Constants Reference

Constants

IBDirection constants

These constants reflect the suggested direction in which a view should grow along the x and y axes.

```
typedef enum {
    IBNoDirection = 0,
    IBMinXDirection = 1,
    IBMaxXDirection = 2,
    IBMinYDirection = 4,
    IBMaxYDirection = 8,
    IBMinXMinYDirection = (IBMinXDirection | IBMinYDirection),
    IBMinXMaxYDirection = (IBMinXDirection | IBMaxYDirection),
    IBMaxXMinYDirection = (IBMaxXDirection | IBMinYDirection),
    IBMaxXMaxYDirection = (IBMaxXDirection | IBMaxYDirection)
} IBDirection;
```

Constants

`IBNoDirection`
Grow the view equally in both directions.

`IBMinXDirection`
Grow the view towards the origin of the containing view along the x axis.

`IBMaxXDirection`
Grow the view away from the origin of the containing view along the x axis.

`IBMinYDirection`
Grow the view towards the origin of the containing view along the y axis.

`IBMaxYDirection`
Grow the view away from the origin of the containing view along the y axis.

`IBMinXMinYDirection`
Grow the view towards the origin of the containing view along both axes.

`IBMinXMaxYDirection`
Grow the view towards the origin of the containing view along the x axis and away from the origin along the y axis.

`IBMaxXMinYDirection`
Grow the view away from the origin of the containing view along the x axis and towards the origin along the y axis.

`IBMaxXMaxYDirection`
Grow the view away from the origin of the containing view along both axes.

Availability
Available in Mac OS X v10.5 and later.

Declared In
`InterfaceBuilderKit/IBGeometry.h`

Document Revision History

This table describes the changes to *Interface Builder Kit Framework Reference*.

Date	Notes
2007-04-18	New collection for the Interface Builder Kit framework.

Document Revision History