Scripting Bridge Programming Guide for Cocoa

Cocoa > Scripting & Automation

2008-03-11

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

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to Scripting Bridge Programming Guide for Cocoa

Introduced in Mac OS X v10.5, Scripting Bridge is a framework and a technology that makes it much easier for Cocoa developers to control and communicate with scriptable applications. Instead of incorporating AppleScript scripts in your application or dealing with the complexities of sending and handling Apple events, you can simply send Objective-C messages to an object that represents an application with a scripting interface. Your Cocoa application can do anything an AppleScript script can, but it does so in Objective-C code that is integrated with the rest of your project's code.

The current version of Scripting Bridge has some limitations that are described in "About Scripting Bridge" (page 9).

Organization of This Document

This guide has the following chapters:

- "About Scripting Bridge" (page 9) describes what Scripting Bridge does and how it does it, and highlights the advantages it brings to Cocoa application development. It also gives an overview of Scripting Bridge classes and methods.
- "Using Scripting Bridge" (page 17) explains how to prepare your project for Scripting Bridge, how to obtain instances of scriptable applications, how to get and set object properties, how to add objects to a scriptable application, and how best to manipulate element arrays.
- "Improving the Performance of Scripting Bridge Code" (page 27) discusses ways in which you can make your Scripting Bridge code more efficient.

See Also

The following documents cover concepts and technologies related to Scripting Bridge:

- *AppleScript Overview*
- *Cocoa Scripting Guide*
- *Ruby and Python Programming Topics for Mac OS X*

About Scripting Bridge

Scripting Bridge is a technology that makes it easy for a program written in Objective-C to communicate with scriptable applications. The following sections explain what Scripting Bridge is and point out its value to Cocoa developers.

The Advantages of Scripting Bridge

Many applications on Mac OS X are scriptable. A scriptable application is one that you can communicate with from a script or another application, enabling you to control the application and exchange data with it. To be scriptable, an application must define an interface for responding to commands. This interface and its implementation must conform to an object model (prescribed by Open Systems Architecture, or OSA) that specifies the classes of scripting objects, the accessible properties of those objects, the inheritance and containment relationships of scripting objects, and the commands that the scriptable application responds to. OSA packages commands to scriptable applications as Apple events and uses the Apple Event Manager to dispatch those events and receive data in return. Apple events are a pervasive feature of Mac OS X; for example, the operating system uses them to control the life cycles of applications and coordinate their interactions.

Note: For a complete description of the concepts and techniques relevant to making Cocoa applications scriptable, see *Cocoa Scripting Guide*.

AppleScript scripts have long been the principal way to communicate with scriptable applications. But an application can also send commands to a scriptable application and receive responses back to it. The "traditional" way of doing this requires you to use the data structures and functions of the OSA framework to create and send Apple events, but this can be a dauntingly complex task. Typical Cocoa applications—until Scripting Bridge, that is—relied on one of two other techniques. They included an AppleScript script in the application bundle and used Cocoa's `NSAppleScript` class to execute the script. Or they used the `NSAppleEventDescriptor` class to create Apple events which they then send with Apple Event Manager routines. However, both of these approaches are not straightforward for Cocoa developers and can be inefficient.

The Scripting Bridge framework makes it possible to send and receive Apple events using Objective-C messages instead of AppleScript commands or Apple-event descriptors. For Cocoa programmers, Scripting Bridge provides a simple and clear model for controlling scriptable applications. It makes Cocoa programs that send and receive Apple events more efficient, and it doesn't require you to have detailed knowledge of the target application's Apple event model. Scripting Bridge integrates well with existing Objective-C code and works with standard Cocoa designs, such as key-value coding, target-action, and declared properties. It has other advantages too.

- It gives your application access to any scriptable feature of any available scriptable application.
- It uses native Cocoa data types—`NSString`, `NSArray`, `NSURL`, and so on—so you'll never have to deal with less familiar Carbon types.

- It manages Apple event data structures for you, using standard Cocoa memory management to allocate and free Apple events.
- It requires much less code than needed to use `NSAppleEventDescriptor` and direct Apple Event Manager calls.
- It checks for syntax errors at compile time, unlike `NSAppleScript`.
- It runs more than twice as fast as a precompiled `NSAppleScript`, and up to 100x as fast as an uncompiled `NSAppleScript`.

Because Scripting Bridge dynamically populates an application's namespace with Objective-C objects representing the items defined by an application's scripting definition, it enables other scripting languages bridged to Objective-C—namely RubyCocoa and PyObjC—to communicate with and control scriptable applications. It thus gives those scripting languages the same advantages as AppleScript scripts and Objective-C programs using Scripting Bridge. For more information on using Scripting Bridge in RubyCocoa and PyObjC code, see *Ruby and Python Programming Topics for Mac OS X*.

Important: The Ruby and Python bridges to Objective-C use framework metadata to learn about non-introspectable types. However, the classes that Scripting Bridge dynamically generates currently do not include any metadata. This mismatch leads to some limitations when using Scripting Bridge in RubyCocoa and PyObjC code. For example, Scripting Bridge declares enumerated types in its Objective-C headers, but these are not bridged to RubyCocoa and PyObjC as symbolic entities. To use an enumerated value in a script you must specify the corresponding integral value; for instance, the value of iChat's 'away' enumerator would be 0x61776179. Another important mismatch to be aware of are Boolean parameters and return types. Scripting Bridge declares these as `BOOL`, which is a `typedef` for `signed char`. However, Ruby evaluates all numbers, even zero, as logically true because they are Number objects. Consequently when using Scripting Bridge in RubyCocoa you must test Boolean values for equality to zero and not whether they are logically false.

These limitations apply to the initial version of Scripting Bridge, which was introduced in Mac OS X version 10.5.

How Scripting Bridge Works

When you request an instance of a scriptable application from Scripting Bridge, it first locates the bundle of the application by bundle identifier, URL, or process identifier (depending on what you specify). It then gets from the bundle the scripting definition of the application contained in its `sdef` file (or, for older applications, its suite terminology and suite definition files) and parses it. From the items in the scripting definition—classes, properties, elements, commands, and so on—it dynamically declares and implements Objective-C classes corresponding to the scripting classes of the application. The dynamically generated classes inherit from one of the classes defined by the Scripting Bridge framework, `SBObject`.

The scripting classes of an application frequently exist in hierarchical relationships of inheritance, but there is also a containment hierarchy, or object graph. When you ask an object for the objects it contains, Scripting Bridge instantiates those objects from the appropriate scripting classes and returns them. If you request the objects contained by one of those objects, Scripting Bridge instantiates new scripting objects from the appropriate class and returns those. Usually this continues until you reach a leaf node of the graph, which is typically a property containing actual data.

At the root of the object graph is the instance of a dynamically generated subclass of `SBApplication` that represents the application object; in the case of iTunes, for example, that subclass is named `iTunesApplication`. For other scripting objects in the object graph (for example, "document" or "track"), Scripting Bridge uses an instance of a dynamically generated `SBObject` subclass; an iTunes track, for example, is represented by an `iTunesTrack` instance. Elements, which specify to-many relationships with other scripting objects, are implemented as methods that return instances of `SBElementArray`. `SBElementArray` is a subclass of `NSMutableArray`, and so you can send it messages such as `addObject:` and `objectAtIndex:`.

Scripting Bridge implements commands and properties (directly or indirectly) as methods of the `SBApplication` subclass or of one of the application's scripting classes. It implements the properties of a class as Objective-C declared properties, which results in the synthesis of accessor methods for those properties. Many of these accessor methods return, and sometimes set, objects of an appropriate Foundation or Application Kit type, such as `NSString`, `NSURL`, `NSColor`, and `NSNumber`. These properties represent the concrete data of a scriptable application. (See "About Scriptable Cocoa Applications" in *Cocoa Scripting Guide* for a mapping of Cocoa types to AppleScript types.) For commands, Scripting Bridge evaluates the direct parameter to determine the class to assign the method implementation to. If it's a specific scripting class (such as `document`), the command becomes a method of that class; if it's of a generic class (such as `specifier`), it becomes a method of the `SBApplication` subclass.

Scripting objects in Scripting Bridge—that is, objects derived from `SBObject`—are essentially object specifiers. That is, they are references to objects in the scriptable application. To get or set concrete data in the application, the scripting object must be evaluated, which results in Scripting Bridge sending an Apple event to the application. For more on scripting objects as object specifiers, see "Object Specifiers and Evaluation" (page 14).

Example: The Sketch Application

The best way to understand how Scripting Bridge processes an application's scripting definition is to look at a "real" application, in this case the Sketch application. The Sketch example project in `<Xcode>/Examples/AppKit` builds a simple but functionally complete graphics application; with it you can draw and color geometric shapes, lines, and text. The project includes an `sdef` XML file that defines the scripting interface of the application. Listing 1-1 shows the part of the `sdef` file that defines the Sketch `document` class.

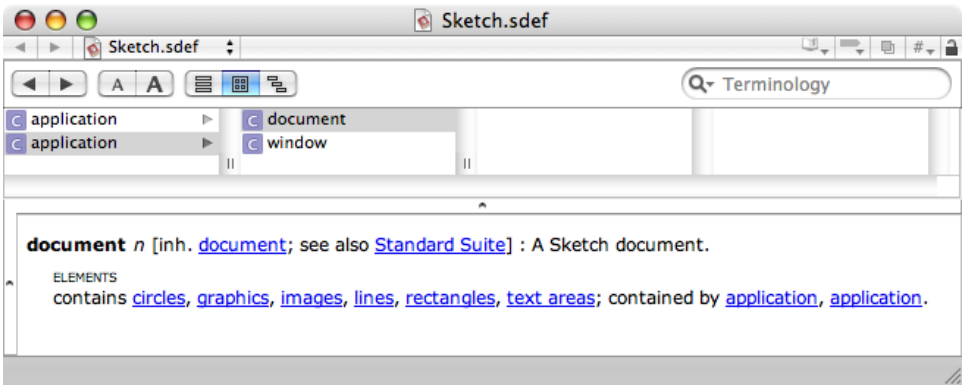
Listing 1-1 The definition of the `document` class for Sketch

```
<class name="document" code="docu" description="A Sketch document.">
            <cocoa class="SKTDrawDocument"/>
            <property name="name" code="pnam" type="text" access="r" description="Its
name.">
                <cocoa key="displayName"/>
             </property>
             <property name="modified" code="imod" type="boolean" access="r"
description="Has it been modified since the last save?">
                <cocoa key="isDocumentEdited"/>
            </property>
            <property name="file" code="file" type="file" access="r" description="Its
location on disk, if it has one.">
                <cocoa key="fileURL"/>
            </property>
```

```
        <!-- This is just here for compatibility with old scripts. New scripts should
 use the more user-friendly file property. -->
            <property name="path" code="ppth" type="text" access="r" description="Its
location on disk, if it has one, as a POSIX path string." hidden="yes">
                <cocoa key="fileName"/>
            </property>

            <element type="graphic"/>
            <element type="box">
                <cocoa key="rectangles"/>
            </element>
            <element type="circle"/>
            <element type="image"/>
            <element type="line"/>
            <element type="text area"/>
            <responds-to name="close">
                <cocoa method="handleCloseScriptCommand:"/>
            </responds-to>
            <responds-to name="print">
                <cocoa method="handlePrintScriptCommand:"/>
            </responds-to>
            <responds-to name="save">
                <cocoa method="handleSaveScriptCommand:"/>
            </responds-to>
        </class>
```

The Dictionary viewer (accessible from the Script Editor application) displays this XML definition of the `document` class as shown in Figure 1-1. It shows the containment of `document` by `application`, and it includes both the common properties of the class (derived from the Standard Suite) as well as the application-specific elements: graphics, boxes, circles, and so on.

Figure 1-1 The `document` class in the Sketch dictionary



Scripting Bridge translates the definition of the Sketch `document` class into the following Objective-C declarations:

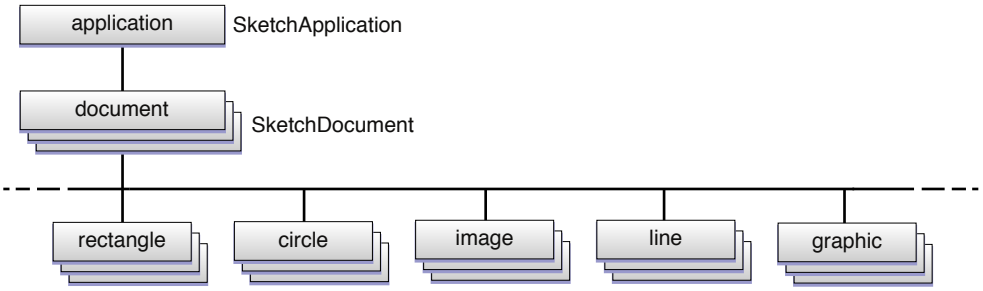
```
@interface SketchDocument : SketchItem

@property (readonly) BOOL modified;  // Has the document been modified since
the last save?
@property (copy) NSString *name;  // The document's name.
```

```
@property (copy) NSString *path;  // The document's path.

@end

// More class declarations here.....

@interface SketchDocument (SketchSuite)

- (SBElementArray *) circles;
- (SBElementArray *) graphics;
- (SBElementArray *) images;
- (SBElementArray *) lines;
- (SBElementArray *) rectangles;
- (SBElementArray *) textAreas;
@end
```

Note that Scripting Bridge makes the declarations for the `SketchDocument` class in two parts. The initial declared properties are for the common properties of the `document` scripting class as defined by the Standard Suite. The category adds declarations specific to the Sketch Suite; these declare methods (corresponding to elements in the scripting definition) that return `SBElementArray` objects.

Figure 1-2 (page 13) depicts how Scripting Bridge converts the Sketch scripting definition of the document class into the implicit graph of objects that constitutes the application's containment hierarchy. At the root of the graph is an instance of the `SketchApplication` class, which inherits from the `SBApplication` class. This application object contains one or more instances of the `SketchDocument` class, which inherits from the `SketchItem` class, itself a descendent of the `SBObject` class. . Each element contained by the `document` class is implemented as a method returning an `SBElementArray` object, which is a subclass of `NSMutableArray`.

Figure 1-2 Object graph representing the Sketch `document` class



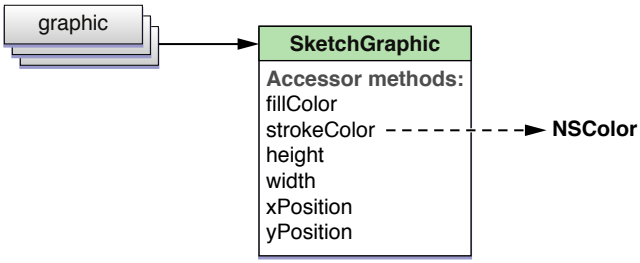
Thus the `graphics` method returns zero or more `SketchGraphic` objects—each corresponds to a `graphic` object in the scripting definition—using an `SBElementArray` object as the container. The `graphic` scripting class of Sketch is particularly interesting because it's also the superclass of the `box` objects, `circle` objects, and the other objects returned by the other element methods. The `graphic` class defines the properties common to all Sketch graphical objects, including fill color, stroke color, origin, and size; these properties are implemented as Objective-C declared properties, which synthesize accessor methods that get and set the property values.

```
@interface SketchGraphic : SketchItem
@property (copy) NSColor *fillColor;  // The fill color.
@property int height;  // The height of the graphic's bounding rectangle.
@property (copy) NSColor *strokeColor;  // The stroke color.
@property int strokeThickness;  // The thickness of the stroke.
```

```
@property int width;  // The width of the graphic's bounding rectangle.
@property int xPosition;  // The x coordinate of the graphic's bounding rectangle.
@property int yPosition;  // The y coordinate of the graphic's bounding rectangle.
@end
```

Figure 1-3 shows both this and the inheritance relationship of the `SketchGraphic` class that Scripting Bridge implements for the `graphic` class.

Figure 1-3 The `graphic` scripting class of Sketch becoming the `SketchGraphic` class



When you set a property in Scripting Bridge, make sure the value of the property is of an appropriate type. Thus, if you try to set the stroke color of a `SketchGraphic` object, you should pass in an `NSColor` object as the new value.

Object Specifiers and Evaluation

Scripting objects in a Scripting Bridge program inherit from the `SBObject` class. `SBObject` objects represent object specifiers in AppleScript. Object specifiers function as proxies for the remote objects in the target application. They help to locate a scriptable object in that application's containment hierarchy. For example, a series of messages sent to objects in a scriptable application could be equivalent to an AppleScript statement such as `get the first track of the third playlist of the source "Library"`. Each segment of this statement, such as `third playlist`, would be represented by an object specifier.

In a Scripting Bridge program, object specifiers are initially in a form of reference supplied by the scripting definition. A `currentTrack` message sent to an `iTunesApplication` instance, for example, would be a reference such as "the 'current track' property of iTunes". When you *evaluate* an object specifier, however, an Apple event is sent to the scriptable application and a more precise, canonical form of reference is obtained for the scripting object. The canonical form of reference provides an exact location of an object within the application, using such things as unique numeric or string identifiers. In the case of `currentTrack`, the canonical reference might be something like "track id 42 of playlist id 23 of source 1".

Because the evaluation of an object specifier results in the dispatch of an Apple event—which is an expensive operation—Scripting Bridge restricts the occasions that trigger evaluation. Lazy evaluation occurs when a message requests concrete data, which is always the value of a property. You can also force evaluation of a reference by sending a `get` message to a scripting object. In both cases, the evaluated object or objects are given a canonical form of reference that doesn't vary over time. (If concrete data is requested, the message also results in a response Apple event that contains the data.)

Often you force evaluation when you want to keep a reference to an object in the scriptable application when the non-canonical form of reference could change at any time. For example, the frontmost document in an application is the referenced object at index zero of the application's `documents` element array. But the ordering of the documents in the array can change as the user clicks on various documents. So to ensure that you have a reference to the current frontmost document, you could do something similar to this:

```
SketchDocument *frontmost_doc = [[[app documents] objectAtIndex:0] get];
```

For more on lazy evaluation and the use of the `get` method, see "Improving the Performance of Scripting Bridge Code" (page 27).

Classes of the Scripting Bridge Framework

The Scripting Bridge framework has three public classes:

- `SBApplication`—A class for objects representing scriptable applications
- `SBElementArray`—A class for collections of objects returned from elements (implemented as methods)
- `SBObject`—The base class for an application's scripting objects

The following class factory methods of `SBApplication` return an object representing an OSA-compliant application (autoreleased in memory-managed environments):

- `+ (id) applicationWithBundleIdentifier:(NSString *)bundleID;`

  Finds the application using its bundle identifier, for example `@"com.apple.iTunes"`. This is the recommended approach for most situations, especially when the application is local, because it dynamically locates the application.

- `+ (id) applicationWithURL:(NSURL *)url;`

  Finds the application using an object representing a URL with a `file:` or `eppc:` file scheme; the latter scheme is used for locating remote applications.

- `+ (id) applicationWithProcessIdentifier:(pid_t)pid;`

  Finds the application using its BSD process identifier (`pid`).

Calling one of these methods is the first step when using Scripting Bridge. Before it returns the application instance, Scripting Bridge parses the scripting definition of the designated application and dynamically implements the required classes. "Creating the Application Object" (page 18) shows how you might use these methods in your Cocoa applications.

> **Note:** There are initializers corresponding to the above class factory methods—for example, `initWithProcessIdentifier:`—which would require you to allocate a generic `SBApplication` object first. But the recommended usage pattern is to call one of the `applicationWith...` methods.

Programs that use Scripting Bridge generally don't need to call many of the methods declared by `SBObject`, `SBApplication`, and `SBElementArray`. The methods you invoke tend to be the ones that Scripting Bridge dynamically declares and implements for an application and its scripting objects. However, several framework methods (listed in Table 1-1) are useful in common situations.

Table 1-1 Commonly called methods of the Scripting Bridge framework

Methods	Purpose
`initWithProperties:` (`SBObject`) `initWithData:` (`SBObject`)	Initializes a scripting object with properties or data; used in conjunction `classForScriptingClass:`. See "Creating and Adding Scripting Objects" (page 23) for an example.
`get` (`SBObject`)	Forces evaluation of a scripting object which is implemented as an object specifier. Evaluation returns the canonical form of reference for the object. For more information, see "Object Specifiers and Evaluation" (page 14) and Listing 3-2 (page 27).
`classForScriptingClass:` (`SBApplication`)	Returns the `Class` object for a scripting class of the application. With this object you can allocate an instance of the scripting class and then initialize it with `initWithProperties:` or `initWithData:`.
`activate` (`SBApplication`) `isRunning` (`SBApplication`)	Allow you to monitor and activate an application. "Improving the Performance of Scripting Bridge Code" (page 27) discusses the use of `isRunning`.
`objectWithName:` (`SBElementArray`) `objectWithID:` (`SBElementArray`) `objectAtLocation:` (`SBElementArray`)	Enables you to obtain a scripting object in an element array by specifying its name, unique identifier, or location. See "Using Element Arrays" (page 21) for an example.
`arrayByApplyingSelector:` (`SBElementArray`) `arrayByApplyingSelector: withObject:` (`SBElementArray`)	Creates an array populated with the objects returned from a message sent to every object in an element array. See "Using Element Arrays" (page 21) for an example.

The Scripting Bridge framework also includes the `SBApplicationDelegate` informal protocol. A delegate may implement the sole method of this protocol (`eventDidFail:withError:`) to handle Apple event errors returned from the target application.

Using Scripting Bridge

Scripting Bridge code is very much like any other Objective-C code. However, there are some differences, mostly deriving from the OSA architecture on which the dispatch and handling of Apple events is based. This chapter describes how to use Scripting Bridge in Objective-C projects, pointing out those differences along the way. "Improving the Performance of Scripting Bridge Code" (page 27) also discusses the correct or optimal use of Scripting Bridge, but from a performance angle.

Preparing to Code

Before you begin writing any Scripting Bridge code for your project, there are a few steps you should complete:

1. Generate header files for all scriptable applications that your code is sending messages to.
2. Add these files to your project.
3. In your header or implementation files, add `#import` statements for the generated header files.
4. Add the Scripting Bridge framework to your project.

You can use `id` to type Scripting Bridge objects dynamically, but such code is vulnerable to programming errors and results in lots of warning messages when you build your project. It is recommended that you generate a header file for each scriptable application, add these files to your project, and in your code give Scripting Bridge specific types based on what you find in the header files. Doing so allows the compiler to check the types of objects (eliminating spurious warning messages) and gives you more assurance that you are sending messages to the correct recipients.

A header file that you generate for a scriptable application serves as reference documentation for the scripting classes of that application. It includes information about the inheritance relationships between classes and the containment relationships between their objects. It also shows how commands, properties, and elements are declared. Taking the iTunes application as an example, the header file shows the definition of the application class (`iTunesApplication`), the application's scripting classes (such as `iTunesTrack` and `iTunesSource`), commands (such as the `eject` method), and properties (such as the `artist` declared property). A header file also includes comments extracted from the scripting definition, such as the comment added to this declaration for the `FinderApplication` class:

```
- (void)empty;   // Empty the trash
```

To create a header file, you need to run two command-line tools—`sdef` and `sdp`—together, with the output from one piped to the other. This is the recommended syntax:

`sdef` *`/path/to/application`*`.app | sdp -fh --basename` *`applicationName`*

The `sdef` utility gets the scripting definition from the designated application; if that application does not contain an `sdef` file, but does instead contain scripting information in an older format (such as the scripting suite and terminology property lists), it translates that information into the sdef format first. The `sdp` tool run with the above options generates an Objective-C header file for the designated scriptable application. Thus, for iTunes, you would run the following command to produce a header file named `iTunes.h`:

```
sdef /Applications/iTunes.app | sdp -fh --basename iTunes
```

Add the generated file to your Xcode project by choosing Add to Project from the Project menu and specifying the file in the ensuing dialog. In any source or header file in your project that references Scripting Bridge objects, insert the appropriate #import statements, such as the following:

```
#import "iTunes.h"
```

Finally, make sure that you have added the Scripting Bridge framework (`/System/Library/Frameworks/ScriptingBridge.framework`) to your project using the Project > Add to Project menu command. It is not necessary to have `#import` statements for the framework because the header files for the scriptable applications do that already.

Creating the Application Object

Before you can send messages to a scriptable application, you need an object that represents the application. As explained in "Classes of the Scripting Bridge Framework" (page 15), the Scripting Bridge framework declares three class factory methods for creating instances of scriptable applications; each takes a different value for locating the application.

- `applicationWithBundleIdentifier:` locates an application by its bundle identifier.
- `applicationWithURL:` locates a (local or remote) application by URL.
- `applicationWithProcessIdentifier:` locates an application by its BSD process identifier (pid).

The recommended method for creating an instance of a scriptable application is `applicationWithBundleIdentifier:`. The method can locate an application on a system even if the user has renamed the application, and it doesn't require you to know where an application is installed in the file system (which could be anywhere). The following line of code creates an instance of the iTunes application:

```
iTunesApplication *iTunes = [SBApplication
applicationWithBundleIdentifier:@"com.apple.iTunes"];
```

If you don't know an application's bundle identifier, you can find it by looking for the value of `CFBundleIdentifier` property in the `Info.plist` file stored in the application bundle.

There might be occasions when using one of the other class factory methods makes sense. For example, if you are writing an application that uses Scripting Bridge for your own personal use, you could refer to applications by URL. The following example creates an instance of the Pages application, locating it by URL in an installation location other than `/Applications`.

```
NSURL *pagesURL = [NSURL fileURLWithPath:[NSString
stringWithFormat:@"%@/Applications/iWork/Pages.app", NSHomeDirectory()]];
PagesApplication *pagesApp = [SBApplication applicationWithURL:pagesURL];
```

Controlling an Application

To control a scriptable application, simply send to the instance of the application or one of its objects a message based on a method declared by the object's class. These methods correspond to commands in the application's scripting definition. The action method listed in Listing 2-1 plays the currently selected iTunes track and then modulates the volume of the sound, eventually restoring it to the original level.

Listing 2-1 Controlling the volume of iTunes

```
- (IBAction)play:(id)sender {
    iTunesApplication *iTunes = [SBApplication
applicationWithBundleIdentifier:@"com.apple.iTunes"];
    if ( [iTunes isRunning] ) {
        int rampVolume, originalVolume;
        originalVolume = [iTunes soundVolume];

        [iTunes setSoundVolume:0];
        [iTunes playOnce:NO];

        for (rampVolume = 0; rampVolume < originalVolume; rampVolume +=
originalVolume / 16) {
            [iTunes setSoundVolume: rampVolume];
            /* pause 1/10th of a second (100,000 microseconds) between
adjustments. */
            usleep(100000);
        }
        [iTunes setSoundVolume:originalVolume];
    }
}
```

Note that this method tests whether the application is running before it attempts to control it. (The `isRunning` method is declared by the `SBApplication` class.) This programming practice is discussed in more detail in "Improving the Performance of Scripting Bridge Code" (page 27).

Getting and Setting Properties

In the object graph that Scripting Bridge dynamically generates for a scriptable application, most objects are containers of other objects (that is, (`SBElementArray` objects) or of objects that refer to another scripting object. In a data-modeling sense, they express to-many or to-one relationships and enable your code to "drill down" the object graph. It is only when you get to the leaf nodes of the graph, which are typically the properties of an object, that you are able to access concrete data, such as a name, a color, or a numerical value. As you may recall, Scripting Bridge implements properties of scripting objects as declared properties in Objective-C.

Getting the value of a property requires you to navigate the object hierarchy of the application until you come to the target object—that is, the object that declares those properties—and then send a message to that object that is the same as the property name. Sometimes you don't have to navigate that far. For example, the fragment of code in Listing 2-2 sends two messages to the `ITunesApplication` object.

Listing 2-2 Getting the name of the current track

```
iTunesApplication *iTunes = [SBApplication
applicationWithBundleIdentifier:@"com.apple.iTunes"];
NSLog(@"Current song is %@", [[iTunes currentTrack] name]);
```

The first message to the application gets the value of its `currentTrack` property; this message yields an object of class `iTunesTrack` representing the track currently playing. This object does itself not represent any concrete data, but the message then sent to it (`name`) returns the value of its `name` property as an `NSString` object.

Important: If you call an accessor method to get a property value of a type descended from `SBObject` (for example, an `iTunesTrack` object), you receive an `SBObject` object even if there is no "real" object set. In a sense, this is an "empty" object that refers to nothing in the target application. Attempting to evaluate the reference results in an error. In such cases, it is recommended that you test the returned object with the `exists` method to determine if it is a reference to a real object.

You can also set the values of properties unless they are (in their declaration) marked as `readonly`. The code in Listing 2-3 implements a command-line tool that clears the `locked` property in items in the Trash.

Listing 2-3 Setting the `locked` property of Finder items

```
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    FinderApplication *theFinder = [SBApplication applicationWithBundleIdentifier:
@"com.apple.finder"];
    SBElementArray *trashItems = [[theFinder trash] items];
    if ([trashItems count] > 0) {
        for (FinderItem *item in trashItems) {
            if ([item locked]==YES)
                [item setLocked:NO];
        }
    }
    [pool drain];
    return 0;
}
```

As the listing shows, you can set the value with a message of the form `set`*Property*`:`, where *Property* is the name of the property with the initial letter capitalized. You can also use dot notation when setting property values—these are, after all, Objective-C declared properties. For example, you could set the value of the `locked` property with this statement:

```
item.locked = NO;
```

However, you could rewrite the Scripting Bridge code in Listing 2-3 (page 20) to be more efficient by using the `setValue:forKey:` method of `NSArray`. The example in Listing 2-4 sends just one Apple event instead of one event per item in the Trash.

Listing 2-4 More efficiently setting the `locked` property

```
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    FinderApplication *theFinder = [SBApplication applicationWithBundleIdentifier:
@"com.apple.finder"];
    SBElementArray *trashItems = [[theFinder trash] items];
```

```
    [trashItems setValue:[NSNumber numberWithBool:NO] forKey:@"locked"];
    [pool drain];
    return 0;
}
```

Note: See "Using Element Arrays" (page 21) for more information on other array-processing approaches and "Improving the Performance of Scripting Bridge Code" (page 27) for a discussion of related performance techniques.

Instead of messages you can use dot notation to traverse the object graph of a scriptable application. The following line is equivalent to the second line in Listing 2-2 (page 20):

```
NSLog(@"Current song is %@", iTunes.currentTrack.name);
```

Sometimes asking a scripting object for the value of a property might not return the expected concrete data. This happens if the returned value itself is an instance of an application-specific subclass of `SBObject`, and thus an object specifier referring to what may be the concrete data. Consider the example in Listing 2-5. This code copies the textual content of each selected Mail message to a TextEdit document.

Listing 2-5 Forcing evaluation of a scripting object to get a property value

```
- (IBAction)copyMailMessage:(id)sender {
    MailApplication *mailApp = [SBApplication
applicationWithBundleIdentifier:@"com.apple.mail"];
    TextEditApplication *textEdit = [SBApplication
applicationWithBundleIdentifier:@"com.apple.TextEdit"];
    SBElementArray *viewers = [mailApp messageViewers];
    for (MailMessageViewer *viewer in viewers) {
        NSArray *selected = [viewer selectedMessages];
        for (MailMessage *message in selected) {
            TextEditDocument *doc = [[[textEdit classForScriptingClass:@"document"]
alloc] init];
            [[textEdit documents] addObject:doc];
            doc.text = [[message content] get];
        }
    }
}
```

The last line of code in the example includes the message `[message content]`, but this message does not yield the textual content of the message as one might expect. Instead, a `get` message is required to force evaluation and thereby obtain the text for assignment. What `[message content]` returns in an instance of `MailText`, which a subclass of `SBObject`. This instance is an object specifier that refers to the actual text. For more on forced evaluation of object specifiers through the `get` method, see "Improving the Performance of Scripting Bridge Code" (page 27).

Using Element Arrays

Element arrays, or `SBElementArray` objects, are collections of scripting objects of the same type. Scripting Bridge implements the elements it finds in an application's scripting definition as methods that return `SBElementArray` instances. Because `SBElementArray` is a subclass of `NSMutableArray`, you can send methods to element arrays that are declared by `NSMutableArray` and its superclass, `NSArray`. For example, as shown in Listing 2-6, you could use an `NSEnumerator` object to enumerate through the array.

Listing 2-6 Using an enumerator to iterate through an element array

```
FinderApplication *theFinder = [[[SBApplication applicationWithBundleIdentifier:
 @"com.apple.finder"] alloc] init];
SBElementArray *trashItems = [[theFinder trash] items];
if ([trashItems count] > 0) {
    NSEnumerator *tren = [trashItems objectEnumerator];
    FinderItem *item;
    while (item = [tren nextObject]) {
        if ([item locked]==YES)
            [item setLocked:NO];
    }
}
```

Or you could even use the fast enumeration feature of Objective-C 2.0 to more efficiently iterate through the array, as shown in Listing 2-7.

Listing 2-7 Using fast enumeration on an element array

```
FinderApplication *theFinder = [[[SBApplication applicationWithBundleIdentifier:
 @"com.apple.finder"] alloc] init];
    SBElementArray *trashItems = [[theFinder trash] items];
    if ([trashItems count] > 0) {
        for (FinderItem *item in trashItems) {
            item.locked = NO;
        }
    }
```

Although it is possible to enumerate through `SBElementArray` objects, as a general rule don't enumerate if you can avoid it. You can get much better performance by using one of the "bulk operation" array methods that Cocoa and Scripting Bridge provide:

- Use `makeObjectsPerformSelector:withObject:` (`NSArray`) to make contained objects perform an operation.
- Use `valueForKey:` and `setValue:forKey:` (NSArray) for bulk fetching and setting, respectively, of the properties of objects in an element array.
- Use `filteredArrayUsingPredicate:` (`NSArray`) to obtain a subset of the objects in the array.
- Use `arrayByApplyingSelector:withObject:` (`SBElementArray`) to get a particular value from each object in the array.

The code in Listing 2-8 uses the last of these methods to create a pop-up list with the calendar names obtained from the iCal application.

Listing 2-8 Using the `arrayByApplyingSelector:` method to populate a pop-up button

```
- (void)awakeFromNib {

    [time setDateValue: [NSDate date]];
    [NSApp setDelegate: self];
    iCalApplication *iCal = [SBApplication
applicationWithBundleIdentifier:@"com.apple.iCal"];
    SBElementArray *calendars = [iCal calendars];
    [cal_popup removeAllItems]; // cal_pop is an outlet to a pop-up button
    [cal_popup addItemsWithTitles:[calendars arrayByApplyingSelector:
@selector(name)]];
```

```
}
```

The code fragment in Listing 2-9 (which is extracted from the method shown in Listing 2-11 (page 24)) uses the `filteredArrayUsingPredicate:` method to create an array containing a subset of calendar events that each have a particular name.

Listing 2-9 Filtering an element array using a predicate

```
iCalEvent *theEvent;
NSArray *matchingEvents = [[theCalendar events] filteredArrayUsingPredicate:
            [NSPredicate predicateWithFormat:@"summary == %@", eventName]];
if ( [matchingEvents count] >= 1 ) {
        // handle any events that match..
}
```

`SBElementArray` also declares methods for extracting individual objects from element arrays using various forms of reference: name (`objectWithName:`), unique identifier (`objectWithName:`), and location within the array (`objectAtLocation:`). The code fragment in Listing 2-10 shows the use of the `objectWithName:` method. (Again, this fragment is taken from the example in Listing 2-11 (page 24).)

Listing 2-10 Getting an object from an element array by specifying its name

```
iCalApplication *iCal = [[[SBApplication
applicationWithBundleIdentifier:@"com.apple.iCal"] alloc] init];
iCalCalendar *theCalendar;

[iCal activate];

NSString *calendarName = [calendar titleOfSelectedItem];

if ( [[[iCal calendars] objectWithName:calendarName] exists] ) {
    theCalendar = [[iCal calendars] objectWithName:calendarName];
}
// etc....
```

Important: If you call an element accessor method (for example, `objectWithName:`) to get an object from an element array, you receive an `SBObject` object even if there is no "real" object with that name, ID, location, or index in the array. In a sense, this is an "empty" object that refers to nothing in the target application. Attempting to evaluate the reference results in an error. In such cases, it is recommended that you test the returned object with the `exists` method to determine if it is a reference to a real object.

For adding scripting objects to element arrays, you may use `NSMutableArray` methods such as `insertObject:atIndex:` and `addObject:`. To remove objects from element arrays, call `removeObject:` (or a related method of `NSMutableArray`) on the element array.

Creating and Adding Scripting Objects

In addition to allowing you to control an application and get and set the properties of scripting objects, Scripting Bridge lets you add scripting objects to an application. For example, you could use Scripting Bridge to dynamically add a playlist to iTunes. There are two important guidelines to remember when adding a scripting object to an application:

- Call `classForScriptingClass:` on the application instance to get the `Class` object to use for allocating the object.

  This method takes the name of the class as specified in the scripting definition—for example, `document`. Do not use a class name in the `sdp`-generated header file as the receiver of the `alloc` method.

- Immediately after creating the object, insert it in the appropriate element array.

  The object is not "viable" in the application until it has been added to its container. Consequently, you cannot set or access its properties until it's been added.

You can use the `init` and `initWithProperties:` methods of `SBObject` to initialize the scripting object. Here are a few lines of code (taken from Listing 2-5 (page 21)) that use `init`:

```
TextEditDocument *doc = [[[textEdit classForScriptingClass:@"document"] alloc]
 init];
[[textEdit documents] addObject:doc];
doc.text = [[message content] get];
```

Note that the code does not set the `text` property until the scripting object has been added to TextEdit's element array of documents. The following code fragment, on the other hand, creates a dictionary with the text property and then initializes the scripting object with it using `initWithProperties::`

```
NSDictionary *props = [NSDictionary dictionaryWithObject:[[message content] get]
 forKey:@"text"];
TextEditDocument *doc = [[[textEdit classForScriptingClass:@"document"] alloc]
 initWithProperties:props];
[[textEdit documents] addObject:doc];
```

Listing 2-11 provides an extended example of conditionally creating and inserting iCal `calendar` and `event` objects, showing both approaches.

Listing 2-11 Creating and adding new scripting objects to the object graph

```
- (IBAction)addUpdateEvent:(id)sender {

    iCalApplication *iCal = [SBApplication
applicationWithBundleIdentifier:@"com.apple.iCal"];
    iCalCalendar *theCalendar;

    [iCal activate];

    NSString *calendarName = [calendar titleOfSelectedItem];

    if ( [[[iCal calendars] objectWithName:calendarName] exists] ) {
        theCalendar = [[iCal calendars] objectWithName:calendarName];
    } else {

        NSDictionary *props = [NSDictionary dictionaryWithObject:calendarName
forKey:@"name"];
        theCalendar = [[[iCal classForScriptingClass:@"calendar"] alloc]
initWithProperties: props] autorelease];
        [[iCal calendars] addObject: theCalendar];
    }

    NSString *eventName = [event stringValue];
    NSDate* startDate = [time dateValue];
```

```
    NSDate* endDate = [[[NSDate alloc] initWithTimeInterval:3600
sinceDate:startDate] autorelease];

    iCalEvent *theEvent;
    NSArray *matchingEvents =
        [[theCalendar events] filteredArrayUsingPredicate:
            [NSPredicate predicateWithFormat:@"summary == %@", eventName]];

    if ( [matchingEvents count] >= 1 ) {
        theEvent = (iCalEvent *) [matchingEvents objectAtIndex:0];
        [theEvent setStartDate:startDate];
        [theEvent setEndDate:endDate];
    } else {
        theEvent = [[[[iCal classForScriptingClass:@"event"] alloc] init]
autorelease];
        [[theCalendar events] addObject: theEvent];
        [theEvent setSummary:eventName];
        [theEvent setStartDate:startDate];
        [theEvent setEndDate:endDate];
    }
    [iCal release];
}
```

Improving the Performance of Scripting Bridge Code

Scripting Bridge code is different from most Objective-C code in that it involves two processes—your process and the scriptable application—and uses Apple events as the way for those processes to communicate. Because of this architecture, performance can be an issue. However, there are several things you can do to optimize performance.

Evaluation of References

Like AppleScript, Scripting Bridge uses Apple events to send and receive information from other applications. However, because sending Apple events can be expensive, Scripting Bridge avoids sending Apple events until it's absolutely necessary or until you request it.

Lazy Evaluation

As described in "Object Specifiers and Evaluation" (page 14), scripting objects are actually object specifiers that are references locating an object in the target application. When you ask for an object from an application, what you actually get back is a reference to it in the terms of the scripting definition; evaluation of the reference sends an Apple event to the application, which returns a more precise, "canonical" reference. Scripting Bridge improves performance through conservative evaluation of references. Normally, it won't evaluate a reference until you need some concrete data from it, which is always the value of an object's property. This is called lazy evaluation.

For example, Scripting Bridge won't send an Apple event when you ask for the first disk of the Finder, but it will send an event when you ask for the name of the first disk of the Finder. Listing 3-1 illustrates this behavior.

Listing 3-1 An example of lazy evaluation

```
- (IBAction)doTest:(id)sender {
    FinderApplication *finder = [SBApplication
applicationWithBundleIdentifier:@"com.apple.finder"];
    SBElementArray *disks = [finder disks];
    FinderDisk *disk = [disks objectAtIndex:0];
    NSString *name = [disk name]; // lazy evaluation occurs
    NSLog(@"Name of first disk is %@", name);
}
```

Most of the time, this lazy evaluation of references won't make much difference to you. However, sometimes you need to be careful to ensure that you get the behavior you expect. Consider the code in Listing 3-2.

Listing 3-2 Effect of elapsed time on lazy evaluation

```
- (IBAction)doTest:(id)sender {
    iTunesApplication *iTunes = [SBApplication
applicationWithBundleIdentifier:@"com.apple.iTunes"];
```

```
    iTunesTrack *savedTrack = [iTunes currentTrack];
    sleep(600);
    NSLog(@"Current track is %@", [savedTrack name]);
}
```

At first glance it might appear that this code logs the name of whatever track was playing 10 minutes ago when it gets to the bottom line. Instead the code logs the name of whatever track is *currently* playing. Why is this so? Recall that Scripting Bridge deals merely with references to objects until you actually need some concrete data from them. So what `savedTrack` stores is a reference to whatever track is currently playing. This reference is evaluated and data is returned only when you call the `name` method, which happens 10 minutes later.

Forcing Evaluation

But what if that's not what you want? What if you want the current track “now"? For this, you need to force the current-track reference to be evaluated as soon as it is created.

To force evaluation you use the `get` method, which is declared by `SBObject`. In effect, the `get` method tells Scripting Bridge "stop being lazy—I want you to evaluate this object now." The following code slightly revises the code in Listing 3-2 to use `get` and thus change the resulting track name:

```
- (IBAction)doTest:(id)sender {
    iTunesApplication *iTunes = [SBApplication
applicationWithBundleIdentifier:@"com.apple.iTunes"];
    iTunesTrack *savedTrack = [[iTunes currentTrack] get];
    sleep(600);
    NSLog(@"Current track is %@", [savedTrack name]);
}
```

Because this code uses the `get` method, `savedTrack` always holds a reference to the track that was playing when the `get` method executed.

Taking Full Advantage of Laziness

Although you may not want Scripting Bridge to be lazy in a particular situation, its lazy evaluation of references dramatically reduces the number of Apple events that need to be sent. Consequently, your application can run significantly faster than it would otherwise. However, if you're not careful about how you write your code, you might needlessly bypass Scripting Bridge's laziness, thus forcing it to send more Apple events than necessary. The following are two common errors:

- **Calling the `get` method when you don’t need to**. Scripting Bridge is excellent at determining when it needs to send an Apple event to get the concrete data you want. When you write `[someObject name]`, for example, Scripting Bridge automatically sends an Apple event to fetch the object's name. If you instead write `[[someObject get] name]`, you force Scripting Bridge to send two Apple events instead of one.
- **Sending the same message multiple times**. Every time you ask an object for the value of a property—such as its name—you force an Apple event to be sent. Thus, in the following example, up to three Apple events might be sent:

  ```
  - (IBAction)doTest: {
      FinderApplication *finder = [SBApplication
  applicationWithBundleIdentifier:@"com.apple.finder"];
  ```

```
    SBElementArray *disks = [finder disks];
    if ([[[disks objectAtIndex:0] name] isEqualToString:@"Macintosh HD"]) { //
 1st AE sent
        NSLog(@"The first disk's name is Macintosh HD");
    } else if ([[[disks objectAtIndex:0] name] isEqualToString:@"Disk 1"]) {
        // If execution reaches here, second Apple event sent
        NSLog(@"The first disk's name is Disk 1");
    } else if ([[[disks objectAtIndex:0] name] isEqualToString:@"My Disk"]) {
        // If execution reaches here, third Apple event sent
        NSLog(@"The first disk's name is My Disk");
    }
}
```

To guarantee that only a single Apple event is sent, call the `name` method only once and store the value in a variable.

```
- (IBAction)doTest: {
    FinderApplication *finder = [SBApplication
applicationWithBundleIdentifier:@"com.apple.finder"];
    SBElementArray *disks = [finder disks];
    NSString *name = [disks objectAtIndex:0];
    if ([name isEqualToString:@"Macintosh HD"]) {
        NSLog(@"The first disk's name is Macintosh HD");
    } else if ([name isEqualToString:@"Disk 1"]) {
        NSLog(@"The first disk's name is Disk 1");
    } else if ([name isEqualToString:@"My Disk"]) {
        NSLog(@"The first disk's name is My Disk");
    }
}
```

Efficiently Enumerating and Filtering Arrays

Scripting Bridge optimizes `SBElementArray` objects as well as `SBObject` objects by making them lazy when it comes to Apple events. They don't send any Apple events until absolutely necessary. When you manually iterate through an `SBElementArray` object, however, you force Scripting Bridge to send as many Apple events as there are items in your array. Take the following example, which makes a list of the name of every disk in the Finder:

```
- (IBAction)doTest:(id)sender {
    FinderApplication *finder = [SBApplication
applicationWithBundleIdentifier:@"com.apple.finder"];
    SBElementArray *disks = [finder disks];
    NSMutableArray *nameArray = [NSMutableArray arrayWithCapacity:[disks count]];
    for (FinderDisk *currentDisk in disks) {
        [nameArray addObject:[currentDisk name]];
    }
```

This code is extremely inefficient. It requires Scripting Bridge to send an Apple event to obtain the number of disks. Then, each time through the fast-enumeration loop, it sends an additional Apple event to get the name of the current disk.

As discussed in "Using Element Arrays" (page 21), whenever possible you should always use one of the "batch operation" array methods instead of enumerating the array. These methods avoid the inefficiency of enumeration because they send a single Apple event rather than one Apple event per item in the array. The methods to use are `makeObjectsPerformSelector:withObject:` and `filteredArrayUsingPredicate:` of `NSArray`, and `arrayByApplyingSelector:` and `arrayByApplyingSelector:withObject:` of `SBElementArray`.

For example, you could rewrite the method above as shown in Listing 3-3.

Listing 3-3 Processing an array with `arrayByApplyingSelector:`

```
- (IBAction)doTest:(id)sender {
    FinderApplication *finder = [SBApplication
applicationWithBundleIdentifier:@"com.apple.finder"];
    SBElementArray *disks = [finder disks];
    NSArray *nameArray = [disks arrayByApplyingSelector:@selector(name)];
    // or you could use valueForKey: (NSArray), e.g.
    // NSArray *nameArray = [disks valueForKey:@"name"];
}
```

This code sends only a single Apple event.

Testing for Launched Applications

If an application isn't executing when Scripting Bridge tries to send it an Apple event, Scripting Bridge may automatically launch it. The launch of the other application may come as a surprise to your users, along with the fact that you application's execution is blocked while the other application is launching.

Because of these potential surprises, it's often a good idea to check whether the target application is running before you try to communicate with a Scripting Bridge message. Suppose you want to get the name of the current track, but only if iTunes is running. You could accomplish by first testing for application execution with the `isRunning` of `SBApplication`, as shown in Listing 3-4.

Listing 3-4 Testing for application execution

```
- (NSString *) nameOfCurrentTrack
{
    // "iTunes" is an instance variable
    if ([iTunes isRunning]) {
        return [[iTunes currentTrack] name];
    }
    return nil;
}
```

Document Revision History

This table describes the changes to *Scripting Bridge Programming Guide for Cocoa*.

Date	Notes
2008-03-11	Changed occurrences of the nonexistent method name +classForApplicationWithBundleIdentifier to the correct method name, +applicationWithBundleIdentifier:.
2007-10-31	New document that describes how to use the Scripting Bridge technology in Cocoa programs to communicate with scriptable applications.

Document Revision History