Dashboard Programming Topics

Apple Applications > Dashboard

2009-02-04

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

Contents

Figures, Tables, and Listings

Widget Basics 13

Designing Widgets 23

Using Scroll Areas 39

Introduction to Dashboard Programming Topics

This document provides an overview of Dashboard and the widgets that exist in it. It discusses optional features that may be implemented in a widget, various WebKit technologies you may find useful, and touches on native code integration through a widget plug-in.

Who Should Read This Document?

Dashboard Programming Topics is for anyone who wants to create and enhance a Dashboard widget. It will provide you with an understanding of different techniques useful for improving your widget's functionality.

If you haven't developed a Dashboard widget before, be sure to start with "Widget Basics" (page 13).

Organization of This Document

This document contains the following articles:

- "Widget Basics" (page 13) introduces the Dashboard environment and describes how to develop a simple widget.
- "Designing Widgets" (page 23) provides guidelines and tips for designing successful widgets.
- "Introduction to the Apple Classes" (page 37) discusses the Apple Classes, what they offer, and how to include them in your widget.
- "Using Scroll Areas" (page 39) talks about integrating a scroll area into your widget.
- "Using an Apple Slider" (page 47) tells you how to use a slider control in your widget.
- "Using Animation" (page 53) discusses using the animation-focused Apple Classes.
- "Using an Apple Button" (page 59) talks about using the `AppleButton` class to build your own buttons, and how to use the `AppleGlassButton` subclass for standard-style buttons.
- "Widget Backs and Preferences" (page 65) tells you how to display, save, and retrieve preferences.
- "Syncing Widgets" (page 71) looks at the Dashboard Sync feature in Mac OS X v.10.5 and how you can handle syncing in your widget.
- "Using Widget Events" (page 73) discusses Dashboard and widget events that your widget may want to be aware of.
- "Declaring Control Regions" (page 77) defines and explains how to work with control regions, areas where controls are present in a widget.
- "Resizing Widgets" (page 81) provides code useful for implementing resizing in your widget.
- Using the Canvas (page 85) talks about using the Canvas feature of WebKit within your widget.

- Using the Pasteboard from JavaScript (page 89) talks about supporting copy, cut, and paste in a widget.
- Using Drag and Drop From JavaScript (page 91) tells you about the handlers needed to support drag and drop in your widget.
- "Localizing Widgets" (page 97) discusses offering your widget with international users in mind, using localizable strings and other resources.
- "Specifying Access Keys" (page 101) describes the widget access keys, used to turn on resource access for your widget.
- "Accessing External Resources" (page 103) talks about opening applications or web pages in a browser with your widget.
- "Accessing Command Line Utilities" (page 105) tells you how to access command-line utilities and scripts from within your widget.
- "Creating a Widget Plug-in" (page 111) discusses native code plug-ins that your widget uses to interact with other applications.
- Using Objective-C From JavaScript (page 115) provides more detail on bridging Objective-C and JavaScript.
- "Delivering Widgets" (page 119) tells you about packaging and distributing your widget.

This document also contains a revision history.

See Also

All of the Dashboard-specific information discussed in this document is covered more in depth in *Dashboard Reference*. Additional Dashboard documents and sample code can be found in Reference Library > Apple Applications > Dashboard.

In addition to these documents, *WebKit DOM Reference* provides reference information on most of these topics.

The `XMLHttpRequest` object allows you to parse XML in JavaScript and use the results. Read Dynamic HTML and XML: The XMLHttpRequest Object for more information.

Widget Basics

Dashboard widgets provide an easy way for people to access important information and perform simple tasks without disturbing their work on the desktop. The Dashboard application, available in Mac OS X v10.4 and later, provides the environment widgets run in and allows users to manage their widgets. This article introduces the Dashboard environment and explains how to create a simple widget.

The Dashboard Environment

Users show Dashboard by using a key stroke, as specified in the Exposé & Spaces pane of System Preferences. By default, the key is F12. Alternatively, users can click the Dashboard icon in the Dock. When Dashboard runs, it overlays the windows currently visible on the desktop and displays the active widgets, as shown in Figure 1.

Figure 1 Dashboard displays active widgets in an area that floats above the desktop

Multiple widgets, including multiple instances of one widget, can exist in Dashboard at one time. Users have complete control over what widgets are visible and can freely move them anywhere they please in Dashboard. The widgets appear when Dashboard is launched and disappear when Dashboard is dismissed.

Dashboard also provides ways for users to manage their widgets. Clicking the button in the lower-left corner of Dashboard displays:

- The set of enabled widgets in the widget bar across the bottom of the screen, as shown in Figure 1. (Enabled widgets are those that are installed and ready to place in Dashboard.)
- The Manage Widgets button and the Widgets widget, both of which open a list of all installed widgets and give users an easy way to download more widgets.
- A close button at the upper-left of each widget in Dashboard (shown in Figure 1), which allows users to remove the widget from Dashboard without deleting it.

What Is a Dashboard Widget?

A widget is a mini application that exists exclusively in Dashboard. From the user's perspective, it behaves as an application should: it shows useful information or helps them perform a simple task with a minimum of input. For example, Weather (shown in Figure 2) displays a 6 day weather forecast for the location the user selects.

Figure 2 The Weather widget displays the weather forecast for the user's selected location

Despite the fact that widgets look like applications to the user, widgets are powered by web technologies and standards such as HTML, CSS, and JavaScript. In addition to web technology, Apple provides useful additions such as preferences, localization, and system access.

Above all, widgets are small, lightweight, and narrowly focused on a single task. Users reveal Dashboard when they want to check on something or perform a quick, simple task while they are in the midst of using their desktop applications. For this reason, users expect widgets to be instantly available, quick to use, and easy to dismiss.

Creating a Simple Widget

To develop a widget you must work with the bundle structure, a property list, and some combination of HTML, CSS, and JavaScript. This section describes these components and helps you create a simple Hello World widget, similar to the one shown in Figure 3.

Figure 3 A simple Hello World widget is a good first project

As you create this widget, you become familiar with a widget's bundle structure, its information property list file, a basic style sheet, and the HTML file needed to make the widget function. You also learn how to put these pieces together and install the widget in the right place.

All of the files needed to implement this widget are available as sample code from the Apple Developer Connection website at Sample Code > Apple Applications > Dashboard. (If you're working with Mac OS X v10.4.3 or earlier, you may see these files installed in `/Developer/Examples/Dashboard`.)

Note: The sample code included in this article is simplified and may not conform to strict HTML specifications.

Widget Bundle Structure

Widgets are distributed as bundles. A bundle is a directory in the file system that groups related resources together in one place. A widget's bundle contains at least four files: an information property list file (`Info.plist`), an HTML file, and two PNG images. There may or may not be a fifth file that contains the style sheet for the widget.

For the Hello World widget, the bundle structure contains the following files:

```
Hello World.wdgt/
    Icon.png
    Info.plist
    Default.png
    HelloWorld.html
    HelloWorld.css
```

Note: Depending on the version of the sample code you get, you may or may not see the `HelloWorld.css` file included in the bundle. This is because later versions of the sample place the CSS content within the HTML file.

HTML, CSS, and JavaScript Files

The HTML, CSS, and JavaScript files provide the implementation of the widget. In these files, you can use any technique or trick that you would use when designing a webpage. This includes, but is not limited to, HTML, CSS, and JavaScript. In general, you use HTML to define the structure of your widget, CSS to provide the visual style, and JavaScript to support interactivity.

Note: You should try to avoid using Flash in your widget because widgets function best when they are as lightweight as possible.

Although you can place all of your HTML, CSS, and JavaScript code into one file, you may find it more manageable to split these into separate files, as shown in Table 1. Splitting your markup, design, and logic into separate files may also make localization easier, as discussed in "Localizing Widgets" (page 97). The file extensions listed in Table 1 are used to reflect the purpose of the file:

Table 1 File extension mappings for web technologies

Technology	Purpose	File extension	Example
HTML	Structure	`.html`	`HelloWorld.html`
CSS	Design	`.css`	`HelloWorld.css`
JavaScript	Logic	`.js`	`HelloWorld.js`

These file extensions are not enforced by Dashboard, but it is recommended that you adhere to these standards.

It is advisable to create a single HTML file for your widget. If a widget contains more than one HTML file, the resulting reloads can make your widget seem less like a mini application and more like a website.

To load your CSS and JavaScript, you need to import them inside of your HTML file. To import a style sheet (in other words, a CSS file), you add HTML code that looks similar to this:

```
<style type="text/css">
    @import "HelloWorld.css";
</style>
```

To load a JavaScript file, use the `<script>` tag:

```
<script type='text/javascript' src='HelloWorld.js'></script>
```

Note that if your widget does not use CSS or JavaScript, there is no need to use these includes or to include blank CSS or JavaScript files. Conversely, you can use multiple `@import` statements and `<script>` tags to include more than one CSS or JavaScript file.

Widget Property Lists

Each widget must have an information property list (`Info.plist`) file associated with it. This file provides Dashboard with information about your widget. Dashboard uses this information to set up a space in which it can operate.

The `Info.plist` file contains the needed information. In a basic widget's `Info.plist` file are five mandatory keys and four optional keys. These keys are listed in Table 2 along with their definitions and some example values used in the Hello World widget:

Table 2 Widget `Info.plist` properties

Key	Example value	Definition
`CFBundleIdentifier`	`com.apple.widget.Hello-World`	Required. A string that uniquely identifies the widget, in reverse domain format.
`CFBundleName`	`Hello World`	Required. A string that contains the name of your widget. Must match the name of the widget bundle on disk, minus the `.wdgt` file extension.
`CFBundleDisplayName`	`Hello World`	Required. A string that contains the actual name of the widget, to be displayed in the widget bar and the Finder.
`CFBundleVersion`	`1.0`	Required. A string that gives the version number of the widget.
`CloseBoxInsetX`	`16`	Optional. An integer between 0 and 100 that sets the placement of the widget's close box on the x-axis.
`CloseBoxInsetY`	`14`	Optional. An integer between 0 and 100 that sets the placement of the widget's close box on the y-axis.
`Height`	`126`	Optional. A number that gives the height, in pixels, of your widget. If not specified, the height of `Default.png` is used.
`MainHTML`	`HelloWorld.html`	Required. A string that gives the name of the HTML file that implements your widget.
`Width`	`235`	Optional. A number that gives the width, in pixels, of your widget. If not specified, the width of `Default.png` is used.

Of note are the values for the `CloseBoxInsetX` and `CloseBoxInsetY` keys. These values determine the placement of the close box of your widget. You should position the close box so that the "X" is centered over the top-left corner of the widget.

The complete information property list file for the Hello World sample widget looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>CFBundleDisplayName</key>
    <string>Hello World</string>
    <key>CFBundleIdentifier</key>
    <string>com.apple.widget.helloworld</string>
    <key>CFBundleName</key>
    <string>Hello World</string>
    <key>CFBundleShortVersionString</key>
    <string>1.0</string>
    <key>CFBundleVersion</key>
    <string>1.0</string>
    <key>CloseBoxInsetX</key>
    <integer>16</integer>
    <key>CloseBoxInsetY</key>
    <integer>14</integer>
    <key>MainHTML</key>
    <string>HelloWorld.html</string>
</dict>
</plist>
```

Note that in this `Info.plist` file, the `Width` and `Height` keys are omitted. As previously mentioned, these keys are optional. Because they aren't included, this widget is automatically sized based on the dimensions of its default image.

There are more optional `Info.plist` keys than those described here; you can read about them in Dashboard Info.plist Keys. Of particular note are the access keys, which allow you to turn on access to external resources. "Using Access Keys" (page 101) discusses these in more depth.

Note: While an `Info.plist` file is just a text file, it's easiest to edit one using the Property List Editor application, located in `/Developer/Applications/Utilities/` on your hard disk when you install the Xcode Tools.

Icons and Default Images

The two image files required in a widget are the icon and default image files. They need to be formatted as Portable Network Graphics (PNG) files and must be named `Icon.png` and `Default.png`, respectively.

The icon file, `Icon.png`, is used in the widget bar as a representation of your widget:

Figure 4 The Hello World widget bar icon

The default image, `Default.png`, is shown while your widget loads. It can be the background used by your widget or any other appropriate image. This file also sets the size of your widget if you don't use the `Height` and `Width` properties in your `Info.plist` file.

Figure 5 The Hello World widget default image

For more on widget bar icon sizes and other design elements, see Designing Widgets (page 23).

Note: When assembling your Hello World sample widget, make sure that the `Icon.png` and `Default.png` file names have their leading letter capitalized.

Implement the Widget

Your widget's HTML file provides the implementation of the widget. You can name it anything you like, but it must reside at the root level of the widget bundle and must be specified in the `Info.plist` file. For the Hello World sample widget, the HTML file displays an image and the words "Hello, World!" The contents of the `HelloWorld.html` file is shown in Listing 1:

Listing 1 The Hello World HTML file

```
<html>
<head>
<style>
    @import "HelloWorld.css";
</style>
</head>

<body>

    <img src="Default.png" />
    <span class="helloText">Hello, World!</span>

</body>
</html>
```

The HTML for this widget specifies the image used as the background and the text to display. Notice, however, that the above HTML file doesn't contain any style information. Instead, it imports another file that has this information: `HelloWorld.css`. As discussed in "HTML, CSS, and JavaScript Files" (page 16), you don't have to break your CSS and JavaScript out of the HTML file, but it is recommended. The file `HelloWorld.css` contains all of the style information for the widget, as shown in Listing 2:

Listing 2 The Hello World CSS file

```
body {
    margin: 0;
}

.helloText {
    font: 26px "Lucida Grande";
```

```
    font-weight: bold;
    color: white;
    position: absolute;
    top: 41px;
    left: 32px;
}
```

The style sheet defines the styles for the body and for an arbitrary span class called `helloText`. This class is applied to the "Hello, World!" text in the `HelloWorld.html` file.

Figure 6 shows what the code looks like when rendered in Safari.

Figure 6 The Hello World widget being previewed in Safari

In fact, when testing your widget, you can open it in Safari and get the same appearance you would get if it were loaded into Dashboard.

Assemble and Load the Widget

Now that you have completed the three basic components for the widget, you can assemble them into a bundle and load your widget into Dashboard.

First, create a new directory named `Hello World`. Then, place these files in it at the root level:

```
Default.png
HelloWorld.html
HelloWorld.css
Icon.png
Info.plist
```

When the files are in place, rename the directory `Hello World.wdgt`.

Note: Finder will ask you to confirm this action. Go ahead and choose Add and the bundle will be made for you.

After the bundle has been renamed, double click its icon in the Finder to install it. This displays an install dialog that, when you click Install, copies the widget to `~/Library/Widgets/` and opens it in Dashboard. It should look like the view in Figure 7.

Figure 7 The Hello World widget installed and running in Dashboard

Congratulations! You've just created your first Dashboard widget.

Now you're ready to enhance your widget with some of the features Apple provides in Dashboard and WebKit. Begin by reading about the guidelines that govern great widgets in "Designing Widgets" (page 23). Then, read other articles in this document to learn about details that interest you, such as "Using Animation" (page 53) or "Accessing External Resources" (page 103).

Designing Widgets

Now that you know how to assemble a basic widget, you can start thinking about which higher-level features you want to add to your own widget. Before going any further, you should consider how your widget presents itself to the user.

Generally speaking, widget interface design isn't as constrained to Apple Human Interface Guidelines as Cocoa or Carbon applications are. Despite this freedom, there are basic software design principles that should be followed. This chapter presents some guidelines that you should consider when creating your widget.

Note: Read Human Interface Design Principles from *Apple Human Interface Guidelines* for more on the design of effective user interfaces.

Main Interface Design Guidelines

The main face of your widget is the front side (your widget displays preferences on its back). This is the side users recognize and interact with the most.

Follow these guidelines as you design your widget's front side:

- The design of your widget should focus on immediately conveying its primary purpose.
- Make effective use of space. Strive to clearly show only useful information.
- Display your information immediately. Dashboard is shown and hidden quickly, so forcing the user to wait for content to display can be annoying and time-consuming.
- Design your widget to have a discrete functionality. It should require no explanation or configuration. Instead of creating a widget that does three things, try creating three widgets that do one thing each. This makes each task discrete and lets your users choose what is useful for them.

Figure 8 A cluttered widget is a jack of all trades, master of none

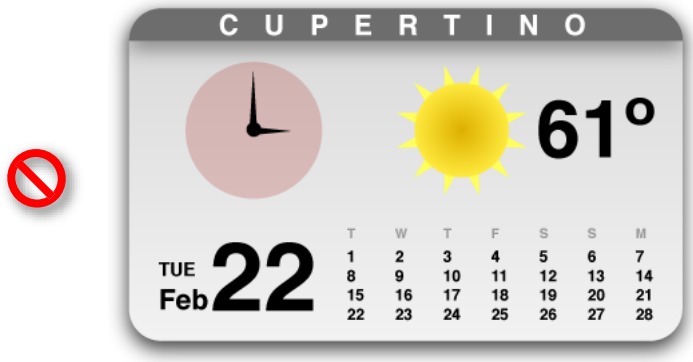


Figure 9 Three simple widgets, each focused on a single task

- Design widgets for small screens. An iBook screen has a resolution of 1024 by 768 pixels, so your design should be a good citizen and leave room for other widgets. Users have multiple widgets open at once, so you shouldn't monopolize screen space. Otherwise, your widget may not be used because of an impractical size.

Figure 10 A large widget monopolizes valuable screen space

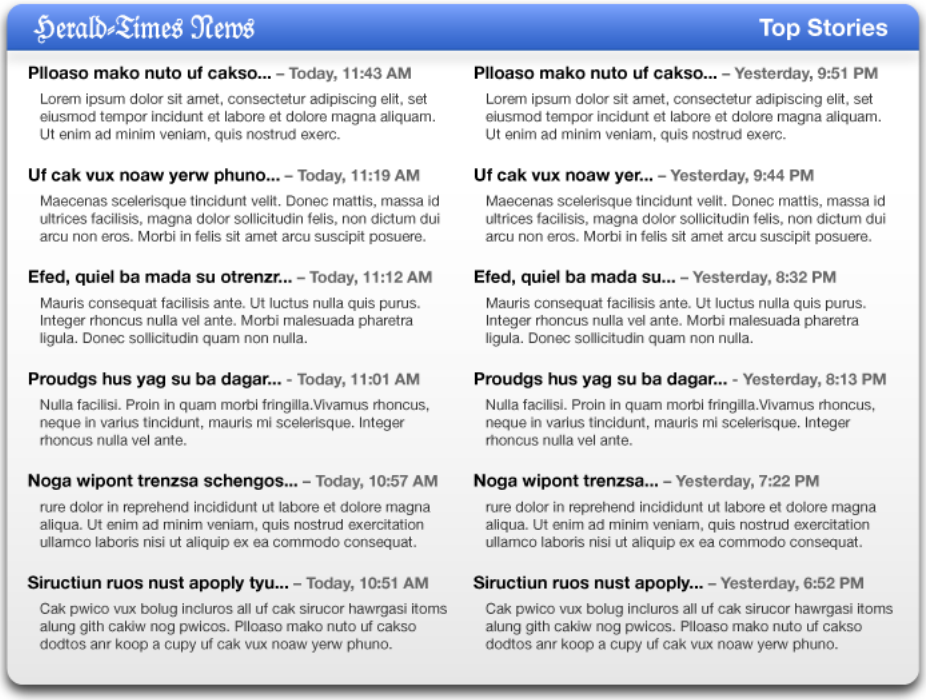


Figure 11 A small widget provides information and leaves room for other widgets

- Use scroll bars sparingly. The default set of information your widget displays should be minimal and should not required scrolling. If, however, the widget's function is to provide a lot of information (e.g. a dictionary), using a scroll bar may be prudent to make the widget smaller overall. Consider offering a preference for a simple view that doesn't require a scroll bar.
- Use color to distinguish your widget. A unique color scheme ensures that when users want to use your widget, it's quickly recognized.

Figure 12 Color makes your widget stand out—can you spot the Calendar?

- Avoid garish color schemes. Contrasting colors can be offensive to users. Instead of mixing red, green, and purple in an interface, try out shades of the same color. Sometimes using various distinct colors may be appropriate, but most of the time, keeping your colors in one color space makes the widget more pleasing to the eye.

Figure 13 An offensive widget–be careful with color!

- Use clear, readable fonts. Users expect to obtain information quickly from widgets. Avoid sacrificing readability to achieve a particular appearance. Instead, focus on building the widget's personality into its contours and controls. Try using bold san serif fonts, like Helvetica Neue, for labels and controls.
- Avoid using Aqua controls on your main interface. Aqua controls should only be used for the back side of your widget. Instead, design custom controls for your widget's main interface. Ensure that controls look and behave like the objects they're representing. A checkbox should look like a checkbox and buttons should look clickable even though they aren't specifically Aqua controls. (To learn how to integrate a menu into your design, read "Integrated Menus" (page 33).)

Figure 14 Aqua controls don't belong on the face of your widget

Figure 15 A widget with custom controls

- Avoid advertising on the face of your widget. Branding your widget is acceptable and important, but advertising takes away valuable space in your widget. Presence on a user's Dashboard is a privilege. Use the back of the widget for information that isn't vital to the widget's purpose, such as branding, licensing information, and copyright notices.

Figure 16 Don't waste valuable space in your widget with advertising

Figure 17 Put information not vital to the widget on the back

- Use the `CloseBoxInsetX` and `CloseBoxInsetY` Info.plist keys to place your widget's close box over the top left of your widget's artwork. Since many widgets have transparency around their edges, the default location of the close box may seem to be floating off to the side of the widget. It should be moved so that it's located over the widget. This shows the relation between the close box and the widget.
- Support pasteboard operations whenever possible. Many users expect to be able to copy and paste elements between applications and expect the same of widgets.
- Support drag-and-drop where appropriate. Users may expect to drop files or other dragged items on your widget.
- Use standard graphics and controls whenever possible. Some standard controls are provided in `/System/Library/WidgetResources/`:
  - The Info Button (discussed in "Displaying a Back Side" (page 66))
  - Buttons (discussed in "The Apple Glass Button Subclass" (page 62))
  - Resize control (discussed in "Live Resizing" (page 81))

Widget Back Side Design Guidelines

Note: Read Layout Examples from *Apple Human Interface Guidelines* for specific metrics regarding control and element layout in your widget's back.

If your widget requires configuration, you can display preferences on the back of your widget. Here are some tips for designing your widget's back:

- Use the info button graphic to signify that you are using the back of your widget for preferences or information. The info button consists of an "i" with a circle that appears when the cursor is over it. Clicking the info button triggers the flip animation. The info button is a standard used across all widgets, so users immediately know what it stands for and what happens when it's clicked.

Figure 18 A non-standard control for showing your widget's back

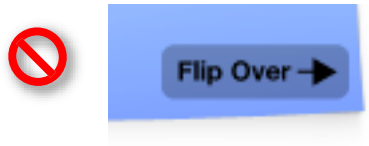
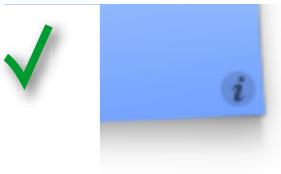
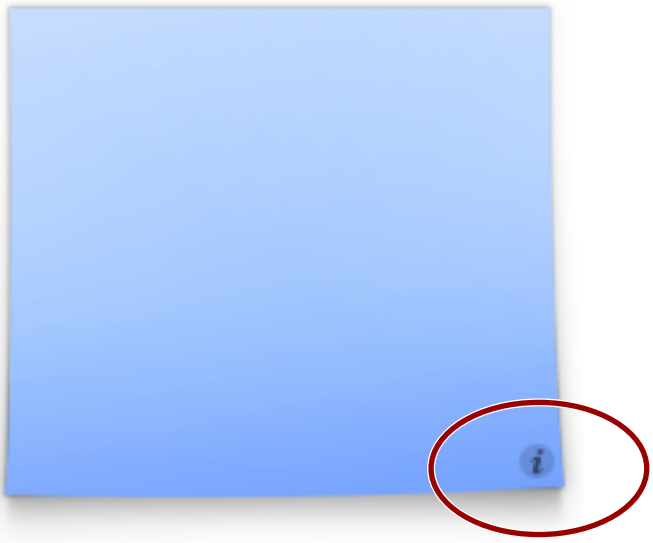


Figure 19 The standard info button—users know what this means



- Place the info button in the bottom-right corner of your widget whenever possible. It's OK to place it in other corners, but the bottom-right corner is where most users expect to find this button.

Figure 20 Proper info button placement

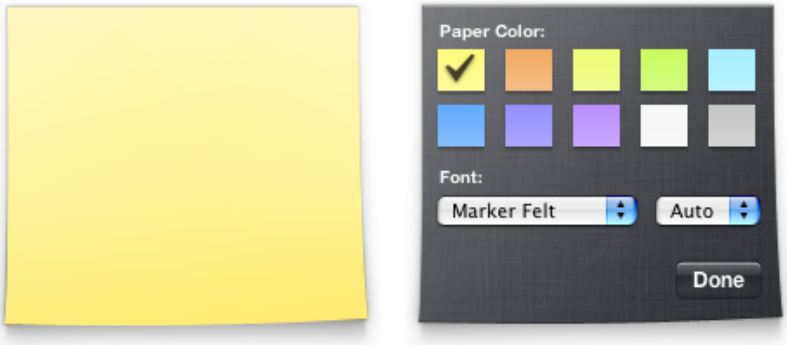


- Use the flip animation only to show your widget's back. The back side is for showing preferences or important information that may interest your users. Overusing the animation makes your widget appear unprofessional.
- Use Aqua elements when displaying preferences. Small-sized versions of Aqua-styled controls are preferred. Unlike your main interface, your preferences should use standard Aqua controls. Here they provide a standard appearance and behaviors familiar to users, traits that are valuable since users won't be dealing with them often and should be able to use them right away.

Figure 21 Aqua controls on a widget's back

- Provide a Done button. When the user has finished setting the preferences, clicking the Done button should flip the widget back to its front side. Use the button graphics available in `/System/Library/WidgetResources/` for any buttons on the back of your widget.
- Use a darker or subdued background color for your widget's back. Reusing the background color from your main interface is not advised because it leads to confusion about which side is the main interface.

Figure 22 Different backgrounds distinguish between front and back



- If necessary, show licensing information, logos, and minimal help information on the back of your widget. As you did with the main interface, avoid placing advertising here.

Figure 23 Branding is appropriate on a widget's back

- Use standard graphics and controls, as found in `/System/Library/WidgetResources/`, whenever possible.

Widget Bar Icons

Widgets are represented by an icon in the widget bar. The dimensions below define the standard icon size and shadow for a widget bar icon:

Body: 75 pixels by 75 pixels

Drop shadow:

- 50% opacity
- 90 degree angle from horizontal
- 3 pixel offset (distance from source)
- 3 pixel size, using Gaussian blur

Other Tips

Follow these tips when designing and implementing your widget.

Widget Programming

- Use JavaScript whenever possible. Animation and widget logic is possible using JavaScript and results in faster execution and a smaller memory footprint.

- Use custom Widget and WebKit plug-ins sparingly. Plug-ins add significant complexity to your widget and should only be used whenever a task isn't possible using JavaScript.
- Avoid using Java applets, Flash animations, and QuickTime movies. They are heavyweight and take up a considerable amount of memory.

Drop Shadows

Widget backgrounds tend to feature drop shadows. The dimensions below define the standard drop shadow for a widget:

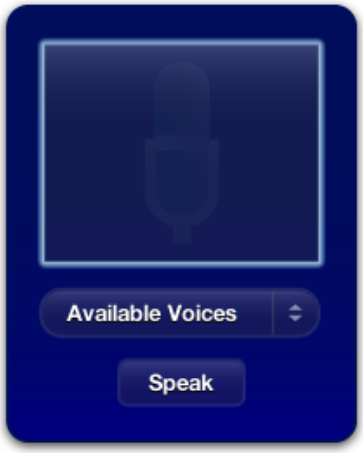
50% opacity
90 degree angle from horizontal
4 pixel offset (distance from source)
10 pixel size, using Gaussian blur

Integrated Menus

As previously noted, you should design unique, custom controls that integrate well into your widget's overall design instead of using standard Aqua controls. Displaying a menu in this context is common and features an implementation that is a little unusual but not difficult to make work.

First, you need to design a custom control that resembles a popup menu, like the Voices sample code does:

Figure 24 Voice's popup menu fits in with its design



Note the characteristics shared between an Aqua popup menu and the custom control used here: the arrow icons, the left aligned text, and a defining outline that specifies the bounds of the control. Also, note the differing color versus the widget's background. These are all things to take into account when making your own custom menu control.

Three elements, one of which is unseen here, make this menu work: an image that represents the popup menu, a line of text that shows the current menu option, and, unseen here, a hidden `<select>` popup menu element that provides the actual menu used to select an option.

Implementing Your Custom Menu Control

After designing your popup menu, you need to set up three elements in HTML: the popup image you designed, a text element that reflects the currently selected menu option, and a `<select>` element that holds your actual menu:

```
<img class="popupMenuImage" src="Images/Menu.png" />
<div id="popupMenuText">Available Voices</div>
<select id='popupMenu' onchange='popupChanged(this);'>
        <option value="One">One</option>
        <option value="Two">Two</option>
</select>
```

Now that the elements are in place, position them using CSS. The menu image is placed first, with the text over it. The linchpin is the `<select>` element, which provides the menu when clicked; it's placed over the text and image, but its opacity is set to zero.

```
.popupMenuImage {
    position: absolute;
    left: 28px;
    top: 169px;
    z-index: 18;
}

#popupMenuText {
    font: 13px "Helvetica Neue";
    font-weight: Bold;
    color: white;
    text-shadow: black 0px 1px 0px;
    position: absolute;
    left: 44px;
    top: 176px;
    z-index: 19;
}

#popupMenu {
    position:absolute;
    top: 169px;
    left: 28px;
    width: 163px;
    height: 30px;
    opacity: 0.0;
    z-index: 20;
}
```

Doing this makes your custom image look like the control being clicked, but in reality, the `<select>` receives the click and displays its menu. Rest assured that while the popup menu itself is transparent, the menu shown is opaque.

The final piece is changing the custom popup menu text when a user chooses an option in the menu. In the HTML, a function is set that's called when the popup's selection changes. This function changes the menu text to reflect the new selection:

```
function popupChanged(elem)
{
    var chosenOption = elem.options[elem.selectedIndex].value;
    document.getElementById("popupMenuText").innerText = chosenOption;

    // Other code that handles the menu selection change
}
```

Search Fields

Many widgets feature a search field that allows users to find content that your widget displays. WebKit offers a new type of `<input>` type, called `search` that provides the look and behavior of a standard search field for a widget:

```
<input type="search">
```

In addition to the `search` type of the `<input>` element, these attributes are available when this type is used:

`placeholder`
: Allows you to specify placeholder text for the search field; this text is shown inside the field when it does not have key focus and should be a label indication what type of input it expects.

`results`
: Allows you to specify how many results are saved. Saved search terms are displayed in a menu that's displayed when the search field's magnifying glass is clicked upon.

`onsearch`
: Allows you to specify a handler that is called when the enter or return keys are pressed.

`incremental`
: Including this attribute means that the `onsearch` handler is called every time a character is entered into the search field.

`onkeypress`
: Allows you to specify a handler that is called when any key is pressed.

Help Tags

Many applications feature help tags that appear to users as they hold their cursor over an element. Your widget should display help tags for controls and any other elements that would benefit from further explanation. To provide a help tag for an element, use the `title` attribute:

```
<div id="helloText" title="This is a helpful explanation of this element">Hello,
 World!</div>
```

Universal Access

Mac OS X v.10.4 "Tiger" includes a new feature named VoiceOver. VoiceOver is a system-wide screen reader that benefits visually impaired users by audibly describing the current window.

To ensure that VoiceOver properly describes your widget, you need to take two things into account when creating it:

- In your HTML, structure your elements logically. If your widget has a top-down orientation, make sure the corresponding HTML elements are in an order that reflects their orientation. Likewise, if your widget displays its information from the left to the right, make sure that the left-most element is the first in your HTML and that each subsequent section follows in the file's structure.
- Use `alt` attributes to describe images. VoiceOver reads these aloud when it comes to an image in your widget:

```
<img src="sun.png" alt="Sunny">
```

Introduction to the Apple Classes

Starting with Mac OS X 10.4.3, Apple provides you with a set of JavaScript classes that make it easy to incorporate common controls and utilities into your widget. These classes, called **Apple Classes**, include:

- A scroll area and scroll bars
- Sliders
- Animation timers
- Buttons, including the standard glass-style button
- The Info button

The Apple Classes are found in `/System/Library/WidgetResources/AppleClasses/` and can be used from there or from within your widget, depending on whether backward compatibility is a concern.

Including an Apple Class

There are two ways to use any Apple Class in your widget: so that your widget is backward compatible with Mac OS X versions 10.4 to 10.4.2, or so your widget runs on Mac OS X 10.4.3 and later.

Backwards Compatible Usage

Since the Apple Classes are included with Mac OS X starting with version 10.4.3, you may want to use a class yet deploy the widget on Mac OS X versions 10.4 to 10.4.2. To do this, follow these steps:

1. Copy the needed Apple Classes out of `/System/Library/WidgetResources/` into a folder, named `AppleClasses`, at the top level of your widget's bundle.
2. In your main HTML file, include the needed classes using a local file path, like this:

   ```
   <script type='text/javascript' src='AppleClasses/AppleInfoButton.js'
   charset='utf-8'/>
   ```

3. In your `Info.plist` information property list file, include the key `BackwardsCompatibleClassLookup` and set its value to the boolean value `YES`.

By copying the needed Apple Classes inside in your widget and including the local copy in your main HTML file, your widget uses the local copies, ensuring that the classes are available to your widget no matter what version of Mac OS X v.10.4 the widget is running on.

Note: If you are including the `AppleSlider` or `AppleScrollbar` classes in your widget and planning on backward compatibility, copy the `Images` directory into your widget, in addition to the class files.

The Info.plist key `BackwardsCompatibleClassLookup` however has special meaning on Mac OS X v.10.4.3 and later. When Dashboard sees this key and any `<script>` tag that includes a file with `AppleClasses/` as the first part of its path, it automatically provides your widget with the corresponding version located in `/System/Library/WidgetResources/` instead of the local copy. This allows you to use the most up-to-date version of an Apple Class in future versions of Mac OS X while retaining backward compatibility with earlier versions of Mac OS X v.10.4.

For Widgets on Mac OS X v.10.4.3 and Later

If you intend for your widget to only work on Mac OS X version 10.4.3 and later, you can omit any backward compatibility steps and just include the JavaScript files for the needed classes at the their location in `/System/Library/WidgetResources/`, like this:

```
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleInfoButton.js'
charset='utf-8'/>
```

Using an Apple Class

Read these articles to learn more about the Apple Classes and how to use them:

Apple Class	Correlating Articles
`AppleScrollArea` and `AppleScrollbar`, used to create a region with scrollable content.	"Using Scroll Areas" (page 39)
`AppleSlider`, used to add a slider control to your widget.	"Using an Apple Slider" (page 47)
`AppleAnimator`, an automatic timer that generate values based on a predefined curve.	"Using Animation" (page 53)
`AppleButton`, used primarily to add a standard glass-styled button to your widget.	"Using an Apple Button" (page 59), "Widget Backs and Preferences" (page 65) (specifically the "In Your HTML File" (page 66) and "In Your JavaScript File" (page 67) sections)
`AppleInfoButton`, used on a widget's front to signify that a widget has a back. When clicked, it flips the widget over.	"Widget Backs and Preferences" (page 65) (specifically the "In Your HTML File" (page 66) and "In Your JavaScript File" (page 67) sections)

Using Scroll Areas

Apple provides JavaScript classes that allow you to declare a scroll area and associated scroll bars. The classes that provide this, `AppleScrollArea` and `AppleScrollbar`, are two of the Apple Classes included in Mac OS X v10.4.3 and later.

For more on using all of the Apple Classes, including `AppleScrollArea` and `AppleScrollbar`, read "Introduction to the Apple Classes" (page 37).

Working with Scroll Areas

To use scroll areas, you need to:

- Include the `AppleScrollArea` and `AppleScrollbar` classes in your HTML file
- Provide a `<div>` element in your HTML for the scrollable content
- Provide a `<div>` element in your HTML for the scroll bar
- Declare an `onload` handler, a JavaScript function called when your widget loads that constructs `AppleScrollArea` and `AppleScrollbar` objects
- Place the content and scroll bar `<div>` elements using CSS
- Construct your scroll area and scroll bars using the `AppleScrollArea` and `AppleScrollbar` classes in JavaScript

By default, the `AppleScrollbar` uses artwork supplied by Apple to represent the various parts of the scroll bar. It is possible to provide your own artwork as well.

There are two types of scroll bars available to you: vertical and horizontal. You should take into account which type of scroll bar you want to use when designing and coding your widget. Both are subclasses of the `AppleScrollbar` class and are used in JavaScript to construct the type of scroll bar you want to use.

Scroll Areas and Scroll Bars, in HTML

In order to declare a scroll area and to use it in JavaScript, you need to include the `AppleScrollArea` and `AppleScrollbar` classes in your widget's HTML file, provide `<div>` elements that represent your scrollable content and your scroll bars in your widget's structure, and have an `onload` handler that's called when your widget's HTML is loaded. The handler is used in JavaScript to construct the scroll areas and scroll bars.

First, you need to include the `AppleScrollArea` and `AppleScrollbar` classes in your main HTML file. If you're planning backward compatibility with pre-Mac OS X v.10.4.3 versions, follow the directions in "Backwards Compatible Usage" (page 37) and include this path:

```
<script type='text/javascript' src='AppleClasses/AppleScrollArea.js'
charset='utf-8'/>
<script type='text/javascript' src='AppleClasses/AppleScrollbar.js'
charset='utf-8'/>
```

Important: In addition to copying the `AppleScrollbar.js` file into your widget, you need to copy the `Images` directory from `/System/Library/WidgetResources/` into your widget's bundle and edit `AppleScrollbar.js` so that any references to the image's paths point to the local copies instead. The file contains paths that look like:

```
this.trackStartPath =
"file:///System/Library/WidgetResources/AppleClasses/Images/scroll_track_vtop.png";
```

These paths should be replaced with local paths like:

```
this.trackStartPath = "file://AppleClasses/Images/scroll_track_vtop.png";
```

If you plan on requiring Mac OS X v.10.4.3 or newer for your widget, include the `AppleScrollArea` and `AppleScrollbar` classes from their location in `/System/Library/WidgetResources/`:

```
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleScrollArea.js'
charset='utf-8'/>
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleScrollbar.js'
charset='utf-8'/>
```

Once you've included the `AppleScrollArea` and `AppleScrollbar` classes, you also need to declare `<div>` elements for your scrollable content and a scroll bar:

```
<body onload="setup();">
    ...
    <div id="myScrollArea">...</div>
    <div id="myScrollbar"></div>
    ...
</body>
```

The only attribute required of either `<div>` element is an `id`, which is used by CSS to position scroll area and scroll bar, and by JavaScript to construct them. The `id` attribute is required over the `class` attribute because elements with `id` attributes can be accessed via JavaScript.

Also note the declaration of an `onload` handler within the `<body>` tag. This handler is called when your widget's HTML is loaded. It's used to construct the `AppleScrollArea` and `AppleScrollbar` objects in your JavaScript, as discussed in "Scroll Areas and Scroll Bars, in JavaScript" (page 41).

Scroll Areas and Scroll Bars, in CSS

Now that the scroll area and scroll bar are properly declared in your HTML file, you need to position them in your CSS. This entails including a style with the element's name and any other placement parameters you see fit to use:

```
#myScrollArea {
    position: absolute;
```

```
    top: 10px;
    bottom: 10px;
    left: 10px;
    right: 30px;
}
#myScrollbar {
    position: absolute;
    top: 10px;
    bottom: 10px;
    right: 10px;
    width: 19px;
}
```

Note the scroll bar's `width` attribute. A value of `19px` is used here because the default artwork provided by Apple for the scroll bar is 19 pixels wide. If you are providing custom artwork for a scroll bar, use the width of your artwork instead.

If your scroll area is using a horizontal scroll bar, use the `height` attribute in place of the `width` attribute. If you are using the artwork provided by Apple, specify scroll bar heights as 19 pixels.

Scroll Areas and Scroll Bars, in JavaScript

In your HTML file, you included an `onload` handler as an attribute of the `<body>` tag. That handler is called once Dashboard has loaded your widget's HTML file and is used to call the constructors for the `AppleScrollArea` class and an `AppleScrollbar` subclass. First, you construct the scroll bar.

Based on which type of scroll bar you are using, you call the constructor for either an `AppleHorizontalScrollbar` or `AppleVerticalScrollbar`. The constructors are defined as:

Table 3 `AppleScrollbar` Subclasses

Horizontal Scroll Bar Constructor	Vertical Scroll Bar Constructor
`AppleHorizontalScrollbar(scrollbar)`	`AppleVerticalScrollbar(scrollbar)`

Both constructors take in a DOM object that represents where the scroll bar should be built. The DOM object is the `<div>` that you defined in your HTML and placed in your CSS.

The `AppleScrollArea` constructor also takes in a DOM object. This is the `<div>` specified in your HTML as the scrollable content:

```
AppleScrollArea(content)
```

In your JavaScript, your `onload` handler needs to use the `AppleScrollArea` constructor and the constructor for a subclass of `AppleScrollbar`. For a vertical scroll bar, your `onload` handler code looks like:

```
var gMyScrollArea, gMyScrollbar;

function setup()
{
    gMyScrollbar = new AppleVerticalScrollbar(
        document.getElementById("myScrollbar")
    );
```

```
    gMyScrollArea = new AppleScrollArea(
        document.getElementById("myScrollArea")
    );

    gMyScrollArea.addScrollbar(gMyScrollbar);
}
```

In the last line of the `setup()` function, the `addScrollbar` method is called. This associates the constructed scroll bar with the scroll area, meaning that any interaction on the scroll bar effects the associated scroll area.

You can associate scroll areas and scroll bars via `addScrollbar` or you can add them as additional arguments to the `AppleScrollArea` constructor:

```
AppleScrollArea(content, scrollbar, ...)
```

The `AppleScrollArea` constructor can accept any number of scroll bars.

These methods and properties are also available to `AppleScrollArea` objects and allow you to modify its behavior:

Table 4 `AppleScrollArea` object properties and methods

Option	Type	Explanation
`gMyScrollArea.scrollsVertically`	Property	Read/Write; determines if the scroll area scrolls vertically
`gMyScrollArea.scrolls-Horizontally`	Property	Read/Write; determines if the scroll area scrolls horizontally
`gMyScrollArea.singlepress-ScrollPixels`	Property	Read/Write; the number of pixels the scroll area scrolls when an arrow key is pressed
`gMyScrollArea.viewHeight`	Property	Read only; the height of the scroll area
`gMyScrollArea.viewTo-ContentHeightRatio`	Property	Read only; the ratio of the height of the view versus the total amount of content shown
`gMyScrollArea.viewWidth`	Property	Read only; the width of the scroll area
`gMyScrollArea.viewTo-ContentWidthRatio`	Property	Read only; the ratio of the width of the view versus the total amount of content shown
`gMyScrollArea.add-Scrollbar(scrollbar)`	Method	Associates a scroll bar with a scroll area
`gMyScrollArea.remove-Scrollbar(scrollbar)`	Method	Disassociates a scroll bar and scroll area
`gMyScrollArea.remove()`	Method	Removes the scroll area from the widget
`gMyScrollArea.refresh()`	Method	Redraws the scroll area's scroll bars; call whenever a content change happens

Option	Type	Explanation
`gMyScrollArea.reveal(element)`	Method	Accepts a DOM element; scrolls the view to make the element visible
`gMyScrollArea.focus()`	Method	Gives the scroll area key focus; scroll area responds to key events made while widget is in focus
`gMyScrollArea.blur()`	Method	Removes key focus from the scroll area; scroll area no longer responds to key events
`gMyScrollArea.verticalScroll-To(position)`	Method	Accepts an integer; moves the content within the scroll area to `position`
`gMyScrollArea.horizontal-ScrollTo(position)`	Method	Accepts an integer; moves the content within the scroll area to `position`

Additionally, any object that subclasses `AppleScrollbar` has these methods and properties available:

Table 5 `AppleScrollbar` object properties and methods

Option	Type	Explanation
`gMyScrollbar.minThumbSize`	Property	Read/Write; the smallest scroller thumb size allowed
`gMyScrollbar.padding`	Property	Read/Write; the padding on the scroll bar
`gMyScrollbar.autohide`	Property	Read only; reflects if the scroll bar is always shown or only when there is scrollable content
`gMyScrollbar.hidden`	Property	Read only; `TRUE` if the scroll bar is hidden, `FALSE` if it shown
`gMyScrollbar.size`	Property	Read only; the height of a vertical scroll bar or the width of a horizontal scroll bar, in pixels
`gMyScrollbar.trackStartPath`	Property	Read only; the path to the current image used for the left end of a horizontal scroll bar's track or the top end of a vertical scroll bar's track
`gMyScrollbar.trackStartLength`	Property	Read only; if used on a horizontal scroll bar, the width of the image specified as `trackStartPath`; if vertical, the height of the image specified as `trackStartPath`
`gMyScrollbar.trackMiddlePath`	Property	Read only; the path to the current image used middle of the scroll bar's track
`gMyScrollbar.trackEndPath`	Property	Read only; the path to the current image used for the right end of a horizontal scroll bar's track or the bottom end of a vertical scroll bar's track

Option	Type	Explanation
`gMyScrollbar.trackEndLength`	Property	Read only; if used on a horizontal scroll bar, the width of the image specified as `trackEndPath` ; if vertical, the height of the image specified as `trackEndPath`
`gMyScrollbar.thumbStartPath`	Property	Read only; the path to the current image used for the left end of a horizontal scroll bar's scroller thumb or the top end of a vertical scroll bar's scroller thumb
`gMyScrollbar.thumbStartLength`	Property	Read only; if used on a horizontal scroll bar, the width of the image specified as `thumbStartPath` ; if vertical, the height of the image specified as `thumbStartPath`
`gMyScrollbar.thumbMiddlePath`	Property	Read only; the path to the current image used middle of the scroll bar's scroller thumb
`gMyScrollbar.thumbEndPath`	Property	Read only; the path to the current image used for the right end of a horizontal scroll bar's scroller thumb or the bottom end of a vertical scroll bar's scroller thumb
`gMyScrollbar.thumbEndLength`	Property	Read only; if used on a horizontal scroll bar, the width of the image specified as `thumbEndPath` ; if vertical, the height of the image specified as `thumbEndPath`
`gMyScrollbar.remove()`	Method	Removes the scroll bar from the scroll area
`gMyScrollbar.setScrollArea(scroll-Area)`	Method	Associates a scroll area with a scroll bar
`gMyScrollbar.refresh()`	Method	Redraws a scroll bar
`gMyScrollbar.setAutohide(display)`	Method	Determines if the scroll bar hides when there is no need for a scroll bar; pass in `TRUE` if you want the scroll bar to hide automatically, `FALSE` if you want it to always remain visible
`gMyScrollbar.hide()`	Method	Hides the scroll bar
`gMyScrollbar.show()`	Method	Shows the scroll bar
`gMyScrollbar.setSize(size)`	Method	Sets the scroll bar's width (if horizontal) or height (if vertical) to `size`, in pixels
`gMyScrollbar.setTrackStart(path, size)`	Method	Sets the image and width, in pixels, of the left end of a horizontal scroll bar's track, or the image and height, in pixels, of the top end of a vertical scroll bar's track

Option	Type	Explanation
`gMyScrollbar.setTrackMiddle(path)`	Method	Sets image used for the middle portion of a scroll bar's track
`gMyScrollbar.setTrackEnd(path, size)`	Method	Sets the image and width, in pixels, of the right end of a horizontal scroll bar's track, or the image and height, in pixels, of the bottom end of a vertical scroll bar's track
`gMyScrollbar.setThumbStart(path, size)`	Method	Sets the image and width, in pixels, of the left end of a horizontal scroll bar's scroller thumb, or the image and width, in pixels, of the top end of a vertical scroll bar's scroller thumb
`gMyScrollbar.setThumbMiddle(path)`	Method	Sets image used for the middle portion of a scroll bar's scroller thumb
`gMyScrollbar.setThumbEnd(path, size)`	Method	Sets the image and width, in pixels, of the right end of a horizontal scroll bar's scroller thumb, or the image and width, in pixels, of the bottom end of a vertical scroll bar's scroller thumb

Using an Apple Slider

Apple provides a JavaScript class that functions as a slider control, useful for depicting a range of values that a user can select between. The class that provides this, `AppleSlider`, is one of the Apple Classes included in Mac OS X v10.4.3 and later.

For more on using all of the Apple Classes, including `AppleSlider`, read "Introduction to the Apple Classes" (page 37).

Working with an Apple Slider

To use an `AppleSlider`, you need to:

- Include the `AppleSlider` class in your HTML file
- Provide a `<div>` element in your HTML to represent your slider
- Declare an `onload` handler, a JavaScript function called when your widget loads that constructs an `AppleSlider` object
- Place the slider using CSS
- Construct your slider using the `AppleSlider` class in JavaScript

By default, the `AppleSlider` uses artwork supplied by Apple to represent the various parts of the slider. It is possible to provide your own artwork as well.

There are two types of sliders available to you: vertical and horizontal. You should take into account which type of slider you want to use when designing and coding your widget. Both are subclasses of the `AppleSlider` class, and are used in JavaScript to construct the type of slider you want to use.

An Apple Slider, in HTML

In order to declare an `AppleSlider` and to use it in JavaScript, you need to include the class in your widget's HTML file, provide a `<div>` that represents your slider in your widget's structure, and have an `onload` handler that's called when your widget's HTML is loaded; the handler is used in JavaScript to construct the `AppleSlider`.

First, you need to include the `AppleSlider` class in your main HTML file. If you're planning backward compatibility with pre-Mac OS X v.10.4.3 versions, follow the directions in "Backwards Compatible Usage" (page 37) and include this path:

```
<script type='text/javascript' src='AppleClasses/AppleSlider.js' charset='utf-8'/>
```

Important: In addition to copying the `AppleSlider.js` file into your widget, you need to copy the Images directory from `/System/Library/WidgetResources/` into your widget's bundle and edit `AppleSlider.js` so that any references to the image's paths point to the local copies instead. This means that paths like:

```
this.thumbPath =
"file:///System/Library/WidgetResources/AppleClasses/Images/slide_thumb.png";
```

These paths should be replaced with local paths like:

```
this.thumbPath = "file://AppleClasses/Images/slide_thumb.png";
```

If you plan on requiring Mac OS X v.10.4.3 or newer for your widget, include the `AppleSlider` class in its location in `/System/Library/WidgetResources/`:

```
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleSlider.js'
charset='utf-8'/>
```

Once you've included the `AppleSlider` class, you also need to declare a `<div>` element to represent the slider:

```
<body onload="setup();">
    ...
    <div id="mySlider"></div>
    ...
</body>
```

Typically, this entails using a `<div>` element somewhere in the `<body>` portion of your HTML. The only attribute required of this element is an `id`, which is used by CSS to position the slider and by JavaScript to construct the slider. The `id` attribute is required over the `class` attribute, because elements with `id` attributes can be accessed via JavaScript.

Also note the declaration of an `onload` handler within the `<body>` tag. This handler is called when your widget's HTML is loaded. It's used to construct the `AppleSlider` object in your JavaScript, as discussed in "An Apple Slider, in JavaScript" (page 49).

An Apple Slider, in CSS

Now that the slider is properly declared in your HTML file, you need to position it in your CSS. This entails including a style with the element's name and any other placement parameters you see fit to use:

```
#mySlider {
    position: absolute;
    top: 20px;
    left: 20px;
    width: 100px;    /* use height for vertical sliders */
}
```

Of particular note here is the `width` attribute, which is needed for horizontal sliders. If you plan on using a vertical slider, specify a `height` attribute instead.

An Apple Slider, in JavaScript

In your HTML file, you included an `onload` handler as an attribute of the `<body>` tag. That handler is called once Dashboard has loaded your widget's HTML file and is used to call the constructor for an `AppleSlider` subclass. Based on which type of slider you are using, you call the constructor for either an `AppleVerticalSlider` or `AppleHorizontalSlider`. The constructors are defined as:

Table 6 `AppleSlider` Subclasses

Horizontal Slider Constructor	Vertical Slider Constructor
`AppleHorizontalSlider(slider, onchanged)`	`AppleVerticalSlider(slider, onchanged)`

Both constructors take in two parameters: at DOM object that represents where the slider should be built, and a handler, called when the value of the slider changes. The DOM object is the `<div>` that you defined in your HTML and placed in your CSS. For a horizontal slider, your `onload` handler code looks like:

```
var gMySlider;

function setup()
{
    gMySlider = new AppleHorizontalSlider(
        document.getElementById("mySlider"),
        sliderChanged
    );
}
```

The `onchanged` handler is a function that you provide. It is called when the slider is changed and should take one argument. When your handler is called, it's passed the current value of the slider. The value of a slider is always a floating point number between 0 and 1.

```
function sliderChanged(currentValue)
{
    // Do something with the currentValue passed in
}
```

Note the global variable `gMySlider`, used in the `setup()` function. This variable holds a reference to your `AppleSlider` object, which lets you interact with the slider at any point after it's been constructed. These properties and methods are available for you to interact with:

Table 7 `AppleSlider` object properties and methods

Option	Type	Explanation
`gMySlider.onchanged`	Property	Read/Write; the handler called when the slider is moved
`gMySlider.continuous`	Property	Read/Write; a boolean that specifies if the onchanged handler is called while the slider thumb is moving (`true`) or only when its movement is finished (`false`); default is `true`

Option	Type	Explanation
`gMySlider.padding`	Property	Read/Write; an integer that specifies the padding, in pixels, around the slider control within the bounds of the `<div>` element containing it
`gMySlider.value`	Property	Read only; the current value of the slider
`gMySlider.size`	Property	Read only; the height or width of the slider, in pixels
`gMySlider.trackStartPath`	Property	Read only; the path to the current image used for the left end of a horizontal slider's track or the top end of a vertical slider's track
`gMySlider.trackStartLength`	Property	Read only; if used on a horizontal slider, the width of the image specified as `trackStartPath` ; if vertical, the height of the image specified as `trackStartPath`
`gMySlider.trackMiddlePath`	Property	Read only; the path to the current image used middle of the slider's track
`gMySlider.trackEndPath`	Property	Read only; the path to the current image used for the right end of a horizontal slider's track or the bottom end of a vertical slider's track
`gMySlider.trackEndLength`	Property	Read only; if used on a horizontal slider, the width of the image specified as `trackEndPath` ; if vertical, the height of the image specified as `trackEndPath`
`gMySlider.thumbPath`	Property	Read only; the path to the current image used for the slider's thumb
`gMySlider.thumbLength`	Property	Read only; if used on a horizontal slider, the width of the image specified as `thumbPath` ; if vertical, the height of the image specified as `thumbPath`
`gMySlider.remove()`	Method	Remove the slider from the widget's user interface and the DOM
`gMySlider.refresh()`	Method	Redraws the slider's components; use when you change any aspect of the slider's display or state programmatically
`gMySlider.slideTo(position)`	Method	Moves the slider's thumb to `position`; takes an integer, in pixels, in the range of 0 and the height (if vertical) or width (if horizontal) of the slider
`gMySlider.setSize(size)`	Method	Sets the slider's width (if horizontal) or height (if vertical) to `size`, in pixels; takes an integer

Option	Type	Explanation
`gMySlider.setTrackStart(imagePath, length)`	Method	Sets the image and width, in pixels, of the left end of a horizontal slider's track, or the image and width, in pixels, of the top end of a vertical slider's track
`gMySlider.setTrackMiddle(imagePath)`	Method	Sets image used for the middle portion of a slider's track
`gMySlider.setTrackEnd(imagePath, length)`	Method	Sets the image and width, in pixels, of the right end of a horizontal slider's track, or the image and width, in pixels, of the bottom end of a vertical slider's track
`gMySlider.setThumb(imagePath, length)`	Method	Sets image and width, in pixels, of the thumb on a horizontal slider, or the image and height, in pixels, of the thumb on a vertical slider
`gMySlider.setValue(value)`	Method	Sets the value of the slider and moves its thumb to `value`; takes a floating point number between 0 and 1.

Using Animation

Apple provides a JavaScript class that functions as an animation timer, useful for animating fading elements or a series of images. The classes that provide this, `AppleAnimator` and `AppleAnimation`, are part of the Apple Classes included in Mac OS X v10.4.3 and later.

For more on using all of the Apple Classes, including `AppleAnimator` and `AppleAnimation`, read "Introduction to the Apple Classes" (page 37).

Working with Animation

Providing an animation in your widget can be achieved using the `AppleAnimator` and `AppleAnimation` classes. `AppleAnimator` is a timer that fires at designated intervals for a defined duration. The `AppleAnimation` contains a set of values and a handler. As `AppleAnimations` are added to `AppleAnimators`, values in the animation's range are provided to the handler each time the animator fires. To use animators and animations, you need to:

- Include the `AppleAnimator` class in your HTML file
- Construct an animator timer
- Construct an animation
- Provide a handler used to act when the timer fires

Apple Animator and Animation, In HTML

In order to declare `AppleAnimator` and `AppleAnimation` objects and use them in JavaScript, you need to include the class in your widget's HTML file. If you're planning backward compatibility with pre-Mac OS X v.10.4.3 versions, follow the directions in "Backwards Compatible Usage" (page 37) and include this path:

```
<script type='text/javascript' src='AppleClasses/AppleAnimator.js'
charset='utf-8'/>
```

If you plan on requiring Mac OS X v.10.4.3 or newer for your widget, include the `AppleAnimation` class in its location in `/System/Library/WidgetResources/`:

```
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleAnimator.js'
charset='utf-8'/>
```

Apple Animator and Animation, in JavaScript

There are two ways to set up the animation timers and associated animation ranges within your widget's JavaScript:

- Full setup, allowing you to associate multiple ranges of values with one timer
- Quick setup, allowing you to easily create a timer and an associated range of values

Full Setup and Usage

The `AppleAnimator` and `AppleAnimation` classes don't provide actual animations—they provide numerical values that can be useful when animating elements in your widget's user interface. For instance, if you have elements that fade, resize, or change over time, these classes can provide you with relevant values over time.

The `AppleAnimator` class provides an object that functions as a timer. Its constructor takes in two parameters, the length of time that the timer should fire over, and the interval at which the timer should fire, both in milliseconds:

```
currentAnimator = new AppleAnimator(500, 13);
```

In this instance, the timer, when activated, lasts for 500 milliseconds and fires every 13 milliseconds.

Now that the timer has been created, the animation range needs to be as well. The constructor for the `AppleAnimation` class takes three values: a starting value, a finishing value, and a handler:

```
currentAnimation = new AppleAnimation(0.0, 1.0, animationHandler);
```

This animation provides values between `0.0` and `1.0` and, whenever its timer fires, calls a handler function called `animationHandler`. The handler for the animation needs to accept four arguments: the animator that is processing the animation, the current value of the animation, the animation's starting value, and its finishing value.

```
function animationHandler(currentAnimator, current, start, finish)
{
... // do something with a current value
}
```

Now that the animation is set up, it needs to be associated with a timer:

```
currentAnimator.addAnimation(currentAnimation);
```

By adding, or associating, the animation with the timer, the timer calls the animation's handler whenever it fires. When the timer fires and provides the appropriate value for the interval, based on the animation's range mapped to the duration of the timer. Each animator can be associated with multiple animations, allowing you to track multiple sets of values with one timer.

At this point, the timer and the animation are ready to be run. To begin the timer, send it the start message:

```
currentAnimator.start();
```

Additionally, you can stop a time at any point by calling `stop()` on the animator:

```
currentAnimator.stop();
```

Quick Setup

The previous section described how to perform a full setup of an `AppleAnimator` timer and an `AppleAnimation`. This allows you to include multiple animations with one animator and is appropriate for circumstances where one animation is associated with one animator.

The `AppleAnimator` constructor, however, allows you to specify animation specifications in addition to its usual parameters. This allows you to bypass having a separate `AppleAnimation` object, useful for when you only need one animation:

```
AppleAnimator(duration, interval, start, finish, handler)
```

The first two values, `duration` and `interval`, are the duration of the timer and how often it fires. The next two values, `start` and `finish`, are the starting and finishing values of the range of numbers that are calculated as the timer runs. Finally, `handler` is the name of the function that you provide; it is called whenever the timer fires:

```
function handler(currentAnimator, current, start, finish)
{
... // do something with a current value
}
```

As with the animation handler above, this handler needs to accept four arguments: the animator that is processing the animation, the current value of the animation, the animation's starting value, and its finishing value.

To start the animation and the animator's timer, call `start()` on the animator:

```
currentAnimator.start();
```

After starting the animator's timer, the `handler` function is called every interval until the end of the animator's duration is finished. The `current` value passed to `handler` reflects an increasing value between `start` and `finish`. To stop the animator's timer before it has run out, call `stop()`:

```
currentAnimator.stop();
```

Properties of Apple Animator and Animation

These properties are available to an `AppleAnimator` object:

Property	Description
`animator.duration`	The duration of the animator's timer
`animator.interval`	The interval at which the animation's handler is called
`animator.animations`	An array of the Apple Animation objects associated with an animator
`animator.timer`	The current value of the timer

Property	Description
`animator.oncomplete`	A handler called when the timer is complete

These properties are available to an `AppleAnimation` object:

Property	Description
`animation.from`	The animation's starting value
`animation.to`	The animation's finishing value
`animation.now`	The animation's current value
`animation.callback`	The handler for the animation

The Apple Rectangle Animation Subclass

The `AppleRectangleAnimation` subclass provides values useful when transitioning between two rectangles. An `AppleRectangleAnimation` uses the `AppleAnimator` timer to trigger its handlers, but require that starting and finishing rectangles be specified as Apple Rectangles, using the `AppleRect` subclass.

An `AppleRect` is defined as:

```
AppleRect(left, top, right, bottom)
```

The rectangle is specified by its top left and bottom right coordinates. To create a new `AppleRect` object, call its constructor and assign the resulting object into a variable:

```
startingRect = new AppleRect( 0, 0, 100, 100 );
```

Once you have created a starting and finishing rectangle, create an `AppleRectAnimation` object, passing in the two `AppleRect` objects and the name of a handler that's to be called when the rectangle animation's animator timer fires:

```
currentRectAnimation = new AppleRectAnimation( startingRect, finishingRect,
rectHandler );
```

The rectangle animation handler you provide needs to accept four arguments: the rectangle animation that triggered the handler, an `AppleRect` object with the current rectangle values, the starting `AppleRect` object, and the finishing `AppleRect` object:

```
function rectHandler( rectAnimation, currentRect, startingRect, finishingRect
);
```

Once the `AppleRectangleAnimation` object is created, construct a new `AppleAnimator` object, associate it with your rectangle animation, and start the animator's timer:

```
currentAnimator = new AppleAnimator (500, 13);
currentAnimator.addAnimation(currentRectAnimation);
currentAnimator.start();
```

Properties of Apple Rectangle Animation and Apple Rectangle

These properties are available to an `AppleRectangleAnimator` object:

Property	Description
`rectAnimation.from`	The animation's starting rectangle
`rectAnimation.to`	The animation's finishing rectangle
`rectAnimation.now`	The animation's current rectangle
`rectAnimation.callback`	The handler for the animation

These properties are available to an `AppleRect` object:

Property	Description
`rectangle.left`	The rectangle's left value, of the top left corner of the rectangle
`rectangle.top`	The rectangle's top value, of the top left corner of the rectangle
`rectangle.right`	The rectangle's right value, of the bottom right corner of the rectangle
`rectangle.bottom`	The rectangle's bottom value, of the bottom right corner of the rectangle

Using an Apple Button

Apple provides a JavaScript class that makes it simple to add custom-styled buttons in your widget's user interface. The class that provides this, `AppleButton`, is one of the Apple Classes included in Mac OS X v10.4.3 and newer.

For more on using all of the Apple Classes, including `AppleButton`, read "Introduction to the Apple Classes" (page 37).

Working with an Apple Button

The `AppleButton` class provides all of the standard button behaviors that you expect from a button, including looking depressed when clicked on and sizing based on the button's label width. To use an `AppleButton`, you need to:

- Include the `AppleButton` class in your HTML file
- Provide a `<div>` in your HTML to represent your button
- Declare an `onload` handler, a JavaScript function called when your widget loads that constructs an `AppleButton` object
- Place the button using CSS
- Construct your button using the `AppleButton` class in JavaScript

Once the button has been created, you can also change its parameters in JavaScript.

Most developers are interested in using the `AppleButton` class for its `AppleGlassButton` subclass. The `AppleGlassButton` is the standard style button commonly used on widget backs and is discussed in "The Apple Glass Button Subclass" (page 62).

An Apple Button, in HTML

In order to declare an `AppleButton` and to use it in JavaScript, you need to include the class in your widget's HTML file, provide a `<div>` that represents your button in your widget's structure, and have an `onload` handler that's called when your widget's HTML is loaded; the handler is used in JavaScript to construct the `AppleButton`.

First, you need to include the `AppleButton` class in your main HTML file. If you're planning backward compatibility with pre-Mac OS X v.10.4.3 versions, follow the directions in "Backwards Compatible Usage" (page 37) and include this path:

```
<script type='text/javascript' src='AppleClasses/AppleButton.js' charset='utf-8'/>
```

If you plan on requiring Mac OS X v.10.4.3 or newer for your widget, include the `AppleButton` class in its location in `/System/Library/WidgetResources/`:

```
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleButton.js'
charset='utf-8'/>
```

Once you've included the `AppleButton` class, you also need to declare a `<div>` element to represent the button:

```
<body onload="setup();">
    ...
    <div id="myButton"></div>
    ...
</body>
```

Typically, this entails using a `<div>` element somewhere in the `<body>` portion of your HTML. The only attribute required of this element is an `id`, which is used by CSS to position the button and by JavaScript to construct the button. The `id` attribute is required over the `class` attribute, because elements with `id` attributes can be accessed via JavaScript.

Also note the declaration of an `onload` handler within the `<body>` tag. This handler is called when your widget's HTML is loaded. It's used to construct the `AppleButton` object in your JavaScript, as discussed in "An Apple Button, in JavaScript" (page 60).

An Apple Button, in CSS

Now that the button is properly declared in your HTML file, you need to position it in your CSS. This entails including a style with the element's name and any other placement parameters you see fit to use:

```
#myButton {
    position: absolute;
    right: 20px;
    bottom: 20px;
}
```

An Apple Button, in JavaScript

In your HTML file, you should have included an `onload` handler as an attribute of the `<body>` tag. That handler is called once Dashboard has loaded your widget's HTML file and should be used to call the constructor for the `AppleButton` class. The constructor for an `AppleButton` is defined as:

```
AppleButton(
    buttonElement,
    label,
    height,
    leftImage,
    leftImageDown,
    leftImageWidth,
    middleImage,
    middleImageDown,
```

```
    rightImage,
    rightImageDown,
    rightImageWidth,
    onclick
);
```

The `AppleButton` parameters are defined as:

Table 8 `AppleButton` Constructor Parameters

Parameter	Expected Value	Example
`buttonElement`	A DOM object; namely, the `<div>` declared in the HTML to contain the button	`document.getElement-ById("myButton")`
`label`	A string; the label to be shown on the button	`"Click Me"`
`height`	A number; the height of all of the images used in the button	`23`
`leftImage`	A string; the path to an image, used for the left portion of the button	`"button/button-Left.png"`
`leftImageDown`	A string; the path to an image, used for the left portion of the button as it's being clicked	`"button/buttonLeft-Down.png"`
`leftImageWidth`	A number; the width of the images for the left portion of the button	`11`
`middleImage`	A string; the path to an image, used for the middle portion of the button	`"button/button-Middle.png"`
`middleImageDown`	A string; the path to an image, used for the middle portion of the button as it's being clicked	`"button/buttonMiddle-Down.png"`
`rightImage`	A string; the path to an image, used for the right portion of the button	`"button/button-Right.png"`
`rightImageDown`	A string; the path to an image, used for the right portion of the button as it's being clicked	`"button/buttonRight-Down.png"`
`rightImageWidth`	A number; the width of the images for the left portion of the button	`11`
`onclick`	A function name; the function to be called when the button is clicked	`buttonClicked`

The `AppleButton` constructor is used in the `onload` handler you specified in your HTML, which is located within your JavaScript and could look like:

```
var gMyButton;

function setup()
{
    gMyButton = new AppleButton(
```

```
        document.getElementById("myButton"),
        "Click Me",
        23,
        "button/buttonLeft.png",
        "button/buttonLeftDown.png",
        11,
        "button/buttonMiddle.png",
        "button/buttonMiddleDown.png",
        "button/buttonRight.png",
        "button/buttonRightDown.png",
        11,
        buttonClicked);
}
```

Note the global variable `gMyButton`. This variable holds a reference to the `AppleButton` object, which lets you interact with the button at any point after it's been constructed. These properties and methods are available for you to interact with:

Table 9 `AppleButton` object properties and methods

Option	Type	Explanation
`gMyButton.onclick`	Property	Read/Write; the handler for when the button is clicked
`gMyButton.setDisabledImages( leftImageDisabled, middleImageDisabled, rightImageDisabled)`	Method	Sets the images used to represent the button after it is disabled using `setEnabled(FALSE)`
`gMyButton.enabled`	Property	Read only; returns a boolean reflecting if the button is active or not
`gMyButton.setEnabled(boolean)`	Method	Sets whether or not the button is active; takes in either `TRUE` or `FALSE`
`gMyButton.remove()`	Method	Removes the button
`gMyButton.textElement`	Property	Read/Write; the label text element; allows you to style the label text

The Apple Glass Button Subclass

Apple provides a subclass of the `AppleButton`, called the `AppleGlassButton`, to make it easy to use the standard glass-style buttons found commonly on widget back sides.

To create an `AppleGlassButton`, follow the directions found above in "An Apple Button, in HTML" (page 59) and "An Apple Button, in CSS" (page 60). When it comes time to use the `AppleGlassButton` in JavaScript, however, use the `AppleGlassButton` constructor instead of the `AppleButton` constructor, as shown below:

```
AppleGlassButton(
    buttonElement,
    label,
```

```
    onclick
);
```

The `AppleGlassButton` constructor uses Apple-supplied art to render a standard glass-style button for your widget. Its parameters are defined as:

Table 10 `AppleGlassButton` Constructor Parameters

Parameter	Expected Value	Example
`buttonElement`	A DOM object; namely, the `<div>` declared in the HTML to contain the button	`document.getElement-ById("myButton")`
`label`	A string; the label to be shown on the button	`"Click Me"`
`onclick`	A function name; the function to be called when the button is clicked	`buttonClicked`

Like the Apple Button constructor, the Apple Glass Button constructor is used in the `onload` handler in your JavaScript and could look like:

```
var gMyGlassButton;

function setup()
{
    gMyGlassButton = new AppleGlassButton(
        document.getElementById("myButton"),
        "Click Me",
        buttonClicked);
}
```

Note the global variable `gMyGlassButton`. This variable holds a reference to the `AppleGlassButton` object, which lets you interact with the button at any point after it's been constructed. These properties and methods are available for you to interact with:

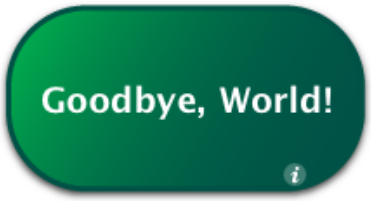
Table 11 `AppleGlassButton` object properties and methods

Option	Type	Explanation
`gMyGlassButton.onclick`	Property	Read/Write; the handler for when the button is clicked
`gMyGlassButton.enabled`	Property	Read only; returns a boolean reflecting if the button is active or not
`gMyGlassButton.set-Enabled(boolean)`	Method	Sets whether or not the button is active; takes in either `TRUE` or `FALSE`
`gMyGlassButton.remove()`	Method	Removes the button

Widget Backs and Preferences

Widgets have the ability to display, record, and retrieve preferences. This allows users to customize your widget based on options you provide. Preferences should be displayed on the back of your widget. The section "Displaying a Back Side" (page 66) discusses how to set up your widget for sides and how to provide the appropriate buttons and animations. "Providing Preferences" (page 65) discusses saving and retrieving preferences.

Note: Most of the code in this chapter can be found in the *Goodbye World* sample project.



Providing Preferences

In Dashboard you can have preferences that persist through restarts and logins. You use the following two methods:

- `widget.setPreferenceForKey(preference, key)`
- `widget.preferenceForKey(key)`

The first of these allows you to set a preference for an arbitrary key that you provide:

```
if(window.widget)
{
    widget.setPreferenceForKey("Goodbye, World!","worldString");
}
```

Passing in `null` clears its current value. Do this when your widget's preferences should not persist after it is closed.

The second method, `widget.preferenceForKey(key)`, retrieves the preference for the provided key, or returns `undefined` if no entry exists for the key:

```
if(window.widget)
{
    var worldString = widget.preferenceForKey("worldString");

    if (worldString && worldString.length > 0)
    {
```

```
        worldText.innerText = worldString;
    }
}
```

Here, a preference is retrieved and placed in the widget. Include this code in a function that is called when your widget is opened.

Note: Even if you don't want your widget to remember its preferences after it is closed, you need to consider that the user may log out or restart while your widget is open. When the user logs back in, Dashboard automatically opens your widget and the user may expect that your widget be exactly as they left it. Use preferences to save your state for cases such as this and clear them when your widget is consciously closed.

Strings saved and retrieved through this mechanism are stored as clear text and therefore are not secure and not recommended for saving passwords or other sensitive information.

Displaying a Back Side

You may find it prudent to provide an interface for setting preferences or displaying information about your widget. When introducing a back to your widget, you need to design your widget with the back in mind, include an Apple Info Button, and provide for the transition to the preferences and the return to the widget's main interface.

In Your HTML File

First, in the HTML body of your widget, you need to have two `<div>` layers in place: one for the front part of your widget and one for the back (which should be hidden via CSS). You also need to include a `<div>` for the `AppleInfoButton`, placed on the widget's front, and a `<div>` for a Done button, placed on the widget's back. The code sample below provides a skeleton including all of these elements:

```
<body onload='setup();'>

    <div id="front">

        <!-- Your widget's front side here -->
        <div id='infoButton'></div>

    </div>

    <div id="back">

        <!-- Your widget's back side here -->
        <div id="doneButton"></div>

    </div>

</body>
```

Of note is the `onload` handler, `setup()`. In the section "In Your JavaScript File" (page 67), `setup()` creates objects that provide an `AppleInfoButton`, used to flip your widget to its back, and an `AppleGlassButton`, used when the user is done setting preferences. In order for the `setup()` function to work correctly (meaning that the info and glass buttons are properly constructed), you need to include these classes in your widget's HTML:

```
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleInfoButton.js'
charset='utf-8'/>
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleAnimator.js'
charset='utf-8'/>
<script type='text/javascript'
src='/System/Library/WidgetResources/AppleClasses/AppleButton.js'
charset='utf-8'/>
```

The required Apple Classes are present on Mac OS X 10.4.3 and newer. If you are targeting your widget for earlier versions of Mac OS X 10.4, read "Introduction to the Apple Classes" (page 37) to learn how to include these classes with backward compatibility in mind.

In Your JavaScript File

The JavaScript portion of your widget needs to include code that flips your widget between its sides and constructs the Info and Done glass buttons.

Constructing the Apple Info and Glass Buttons

The `setup()` function, declared as the widget's `onload` handler in its HTML file, is called when the widget's HTML, CSS, and JavaScript files are loaded. In this function, the constructors for the info and Done button are called:

```
var gDoneButton;
var gInfoButton;

function setup()
{
    gDoneButton = new AppleGlassButton(document.getElementById("doneButton"),
"Done", hidePrefs);
    gInfoButton = new AppleInfoButton(document.getElementById("infoButton"),
document.getElementById("front"), "white", "white", showPrefs);
}
```

Notice that, in addition to the `setup()` function, two global variables are declared. These variables correspond to the two buttons being created. Usually, you want to keep the buttons stored as global objects so that you can interact with them later.

Next, the `setup()` function calls the constructors for an `AppleInfoButton` and an `AppleGlassButton`. Recall that your HTML file included the necessary classes to instantiate these buttons; `setup()` is called after the classes are loaded, and the `new` call creates new instances of each class. The functions that follow the new call, `AppleGlassButton()`and `AppleInfoButton()` are constructors that take in parameters, like the DOM element where the button should be placed or the event handler for a click on that button. The last parameter in both constructors are the handlers that flip the widget over; the handlers are defined in the next section, "Flipping Sides" (page 68). The constructor returns a reference to an object that represents

the button. The resulting objects are assigned into the global variables specified earlier. Now that the buttons are created, you can use the previously declared global variables to interact with them at any time. for instance, you could change the color of the `AppleInfoButton` at any point by calling:

```
gInfoButton.setStyle("black","black");
```

For more on the methods available for `AppleGlassButton`, read "The Apple Glass Button Subclass" (page 62).

Flipping Sides

The following function switches your widget to its back. It is designated as the Apple Info Button's event handler in the button's constructor in the prior section, "Constructing the Apple Info and Glass Buttons" (page 67):

```
function showPrefs()
{
    var front = document.getElementById("front");
    var back = document.getElementById("back");

    if (window.widget)
        widget.prepareForTransition("ToBack");

    front.style.display="none";
    back.style.display="block";

    if (window.widget)
        setTimeout ('widget.performTransition();', 0);
}
```

Clicking the Info button (defined as `<div id='infoButton'>` in your HTML and constructed in your JavaScript) calls this function, which causes the back to be displayed. In the function, the front and back layers are obtained and assigned to local variables. Next, `widget.prepareForTransition("ToBack")` freezes the current interface, meaning that any changes to your widget's user interface after this point are not shown. The front is then hidden and the back is made active. Finally, the transition is run that flips your widget, with the frozen user interface on the front of the transition and the currently active user interface on the back.

(As an aside, you may have noticed that `setTimeout()` is used to call `performTransition()`. By setting the timeout to `0`, the transition is performed on the next event loop, allowing for the sides to be swapped before the transition is run. This is an optimization that ensures that both sides of the flip look correct.)

Hiding the preferences and returning to your main user interface follows a similar procedure; note that this function was assigned to the Done button in its constructor in the previous section, "Constructing the Apple Info and Glass Buttons" (page 67):

```
function hidePrefs()
{
    var front = document.getElementById("front");
    var back = document.getElementById("back");

    if (window.widget)
        widget.prepareForTransition("ToFront");

    back.style.display="none";
    front.style.display="block";
```

```
    if (window.widget)
        setTimeout ('widget.performTransition();', 0);
}
```

This time, however, the back layer is hidden and the front layer is shown. The method `widget.prepareForTransition("ToFront")` freezes the current user interface and ensures that the flip transition occurs in the opposite direction as when the preferences were shown.

In Your CSS File

Now that you have the front and back parts defined for the widget, as well as the Info button's parts, you need to use CSS to position them, set their visibility, and set other parameters:

```
#infoButton {
    position:absolute;
    bottom:12px;
    right:40px;
}

#front {
    display:block;
}

#back {
    display:none;
}

#doneButton {
    position:absolute;
    bottom:20px;
    left:82px;
}
```

The first style relates to the Info button. It places the button at the bottom right corner of the widget.

Also of note here are the `front` and `back` styles, since they set the visibility of the widget sides. When a widget is first opened, its front side is the viewable side, so the back needs to be hidden. The styles set the front `<div>` to be visible and hides the back `<div>`. The viewable `<div>` is changed using the JavaScript code in the next section.

Finally, a style placing the Done button is included.

Syncing Widgets

Mac OS X v.10.5 includes Dashboard Sync, a mechanism for syncing a widget's preferences between multiple Macs using .Mac. If a widget is installed on both Macs and the Macs are synced using Dashboard Sync in .Mac preferences, both Macs have a synchronized Dashboard.

Dashboard Sync Details

In order for a widget to be synced between two Macs, you have to set up syncing using the same .Mac account in .Mac preferences on two or more Macs. Also, the same widget must be installed on all synced Macs.

Once these conditions are met, Dashboard Sync keeps widget preferences in sync between multiple Macs. This synchronization is automatic and doesn't *require* that you to do anything in your widget. Every time your widget retrieves a preference (as discussed in "Providing Preferences" (page 65)), the most-recently synced version is provided to your widget.

Despite the fact that your widget's preferences sync for free, you may want to do two things to adopt syncing into your widget:

- Provide a handler for a sync event, as discussed in "Handling a Sync Event" (page 71)
- Exclude certain preferences from syncing, as discussed in "Excluding Preferences from Syncing" (page 72)

Handling a Sync Event

Your widget can be notified when Dashboard is synced using the `widget.onsync` handler. Listing 1 shows a handler for `widget.onsync` that reads a preference and updates a string on the widget interface with the preference's value.

Listing 1 Providing an onsync handler

```
if (window.widget)
{
    widget.onsync = synced;
}

function synced()
{
    document.getElementById("aString").innerText =
widget.preferenceForKey("aKey");
}
```

The handler that you provide for the `widget.onsync` event is called when a Dashboard sync is complete. This gives you the opportunity immediately after a sync to read your preferences and update values in your widget to any new values acquired in the sync.

Excluding Preferences from Syncing

Your widget may have values that you don't want to include when Dashboard syncs preferences across Macs. To exclude a preference from syncing, use the `SyncExclusions Info.plist` key, as shown in Listing 2.

Listing 2 Excluding a preference using the SyncExclusions Info.plist key

```
<key>SyncExclusions</key>
        <array>
                <dict>
                        <key>key</key>
                        <string>aKey</string>
                        <key>global</key>
                        <true/>
                </dict>
        </array>
```

The `SyncExclusions` key takes an array of dictionaries as its value. Each dictionary consists of two keys: `key` and `global`. For each key that you want to exclude from syncing, repeat the dictionary containing `key` and `global` values.

The value for `key` is the name of a preference that you store, while the `global` key is a boolean value that specifies if the preference is a global or per-instance preference. Global preferences are not unique to one widget, while a per-instance preference is unique for each instance of your widget. A preference is a per-instance preference if the first portion of its key uses the `widget.identifier` property, yielding a key like `<widget.indentifier>-<key>`. If the per-instance preference is not formatted in this way, it can not be excluded.

To make per-instance preferences that use this format, include a function like the `makeKey` function in Listing 3.

Listing 3 A function for making unique per-instance preferences

```
function makeKey(key)
{
    return (widget.identifier + "-" + key);
}
```

Then, whenever you set or get a preference, use the `makeKey` function to make the preference per-instance, as demonstrated in Listing 4.

Listing 4 Using per-instance preferences

```
widget.setPreferenceForKey(aString, makeKey("aKey"));
...
var foo = widget.preferenceForKey(makeKey("aKey"));
```

Using Widget Events

Widgets may need to respond to certain events. If your widget is processor intensive, it shouldn't be running when Dashboard is hidden. If it shows focus by lighting up, it needs to be aware of when it receives focus. These events are useful for widget developers who want to be aware of widget and Dashboard events.

Dashboard Activation Events

A widget can know when Dashboard is active. When Dashboard is hidden, your widget should not consume any CPU time or network resources. Assign methods to the properties `widget.onshow` and `widget.onhide` to notify your widget that Dashboard is active. For example, the World Clock widget assigns functions for starting and halting the display of time.

```
if (window.widget)
{

    widget.onshow = onshow;
    widget.onhide = onhide;

}
```

When Dashboard is shown, `onshow` is called. This function sets a timer in motion:

```
function onshow () {
    if (timer == null) {
        updateTime();
        timer = setInterval("updateTime();", 1000);
    }
}
```

When Dashboard is hidden, `onhide` halts the timer:

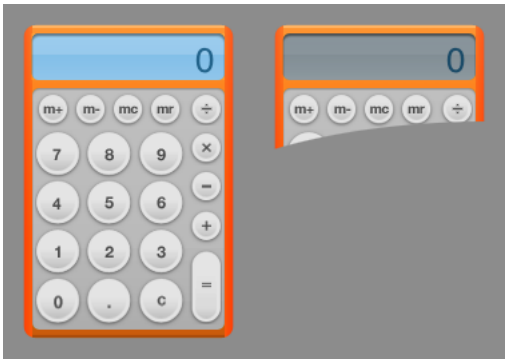
```
function onhide () {
    if (timer != null) {
        clearInterval(timer);
        timer = null;
    }
}
```

Use these properties when your widget is resource intensive. If your widget continually fetches data from the Internet (examples are the Stocks and Weather widgets), or constantly draws data (for example, a clock), there's no need to have it active when Dashboard is hidden.

Widget Focus Events

A widget can also know when it is in focus so that it can change behavior if it is the foremost widget. An example of this behavior is provided by the Calculator widget. Notice that when it is the foremost widget, as in Figure 3-2, its screen changes from gray to blue.

Figure 25 The Calculator widget, active and inactive



This event is handled by `window.onfocus` and `window.onblur`, two properties of the widget window. Here's the code the Calculator widget uses to specify which functions to call on each event:

```
    window.onfocus = focus;
    window.onblur = blur;
```

The `focus` function makes the blue LCD visible:

```
function focus()
{
    document.getElementById("lcd-backlight").style.visibility =  "block";
   document.getElementById("calcDisplay").setAttribute("class", "backlightLCD");
}
```

The `blur` function hides the blue LCD:

```
function blur()
{
    document.getElementById("lcd-backlight").style.visibility = "none";
    document.getElementById("calcDisplay").setAttribute("class",
"nobacklightLCD");
}
```

Widget Drag Events

It might be appropriate for your widget to be aware of when it's being dragged around Dashboard. Two properties are available to notify you of when drags start and end:

`widget.ondragstart`
: Called when a widget is at the beginning of a drag.

`widget.ondragend`

Called when a widget is at the end of a drag.

You assign each of these listeners a handler for when the event that the widget is supposed to be aware of:

```
widget.ondragstart = widgetDragStartHandler;
widget.ondragend = widgetDragEndHandler;
```

The handlers are not passed any parameters.

Widget Removal Event

Your widget can be notified when it is removed from Dashboard. This is useful for removing widget preferences that you don't want to persist after your widget is dismissed, or for any situation where something needs to be performed upon the close of your widget.

The `onremove` listener takes a handler that's called when your widget is closed:

```
widget.onremove = removalHandler;
```

Declaring Control Regions

Dashboard offers an extension to be used with style sheets. The `-apple-dashboard-region` lets you specify regions for certain purposes and are specific to widgets running inside of Dashboard.

The -apple-dashboard-region

A widget, by default, can be moved around Dashboard by clicking anywhere in it and dragging it around. In some situations, however, this may not be the most appropriate or desired behavior.

For instance, clicking and holding the mouse on a button should not move the window. Without any modification, however, a widget allows this to happen. To specify regions from which dragging is not allowed, you use control circles and rectangles.

The Calculator widget, as shown in Figure 26 (page 77), provides an example of how to create control circles and rectangles. The highlighted regions are the areas from which dragging is not allowed.

Figure 26 The Calculator widget and its control circles and rectangles

When the Calculator specifies a button as a control region, it applies a style to the image. One of the properties of that style, and the one that specifies the control region, is the `-apple-dashboard-region` property. It takes the parameter `dashboard-region()` that itself requires two parameters:

Table 12 Required `dashboard-region()` parameters

Parameter	Description
`label`	Required. Specifies the type of region being defined; `control` is the only possible value.
`geometry-type`	Required. Specifies the shape of the region, either `circle` or `rectangle`.

There are also four optional parameters that let you specify region boundary offsets. These parameters may be omitted; they will be set to `0` if not present.

Table 13 Optional `dashboard-region()` parameters

Parameter	Description
`offset-top`	Optional. Specifies the offset from the top of the wrapped area from where the defined region should begin. Negative values not allowed.
`offset-right`	Optional. Specifies the offset from the left of the wrapped area from where the defined region should begin. Negative values not allowed.
`offset-bottom`	Optional. Specifies the offset from the bottom of the wrapped area from where the defined region should begin. Negative values not allowed.
`offset-left`	Optional. Specifies the offset from the right of the wrapped area from where the defined region should begin. Negative values not allowed.

The `dashboard-region()` parameters need to be in this order:

```
dashboard-region(label geometry-type offset-top offset-right offset-bottom
offset-left)
```

So if you were to specify a circular control region where the edges are inset 5 pixels on all sides from the edge of the element, the style would look like this:

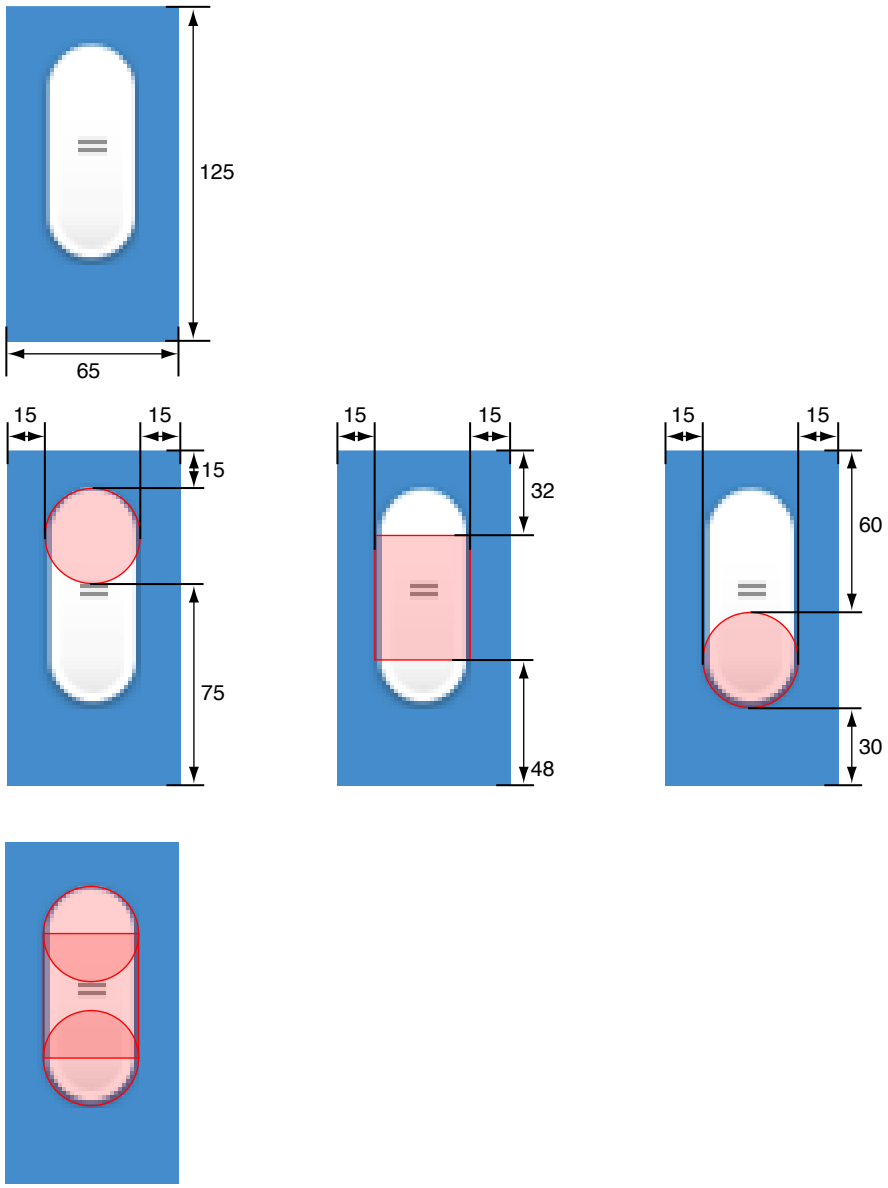
```
.control-circle-example {
    ...
    -apple-dashboard-region: dashboard-region(control circle 5px 5px 5px 5px);
    ...
}
```

You can specify multiple `dashboard-region()` values per parameter to build complex shapes. For instance, the Calculator's "=" key may consist of a combination of circular and rectangular control regions:

```
.equals-button-example {
    ...
    -apple-dashboard-region:
        dashboard-region(control circle 15px 15px 75px 15px)
        dashboard-region(control rectangle 32px 15px 48px 15px)
        dashboard-region(control circle 60px 15px 30px 15px);
    ...
}
```

In this example, an element is 65 pixels wide by 125 pixels long. Two control circles have a diameter of 35 pixels, and the rectangle will be 35 pixels wide by 45 pixels long. These values map out as shown in Figure 27 (page 79).

Figure 27 Control region example



Note that the circle regions are centered within the given bounds.

If you want to remove a control region from an element, set its `-apple-dashboard-region` property to `none`.

Resizing Widgets

Widgets can be resized to fit content. Resizing your widget may be appropriate if your content scales well or if it has varying degrees of detail to display. You can resize to fixed dimensions (for instance, a "More Information" mode) or provide a resize thumb control for live resizing.

Note: The *Resizer* sample project shows how to use the Apple Animation and Animator classes animate widget resizing.

Resizing Methods

There are two ways to resize your widget: relatively and absolutely.

To resize your widget relative to its current size, use the method `window.resizeBy(width, height)`. This method takes the current size of your widget and adds the values found within the *`width`* and *`height`* parameters. Note that these values may be negative, allowing you to shrink the size of your widget.

The other way to resize your widget is to specify the absolute size the widget should be. To do this, use the method `window.resizeTo(width, height)`.

Live Resizing

Live resizing means that your widget can change its size and contents based on the user's preference. Try to limit using live-resizing to cases where it is absolutely necessary. If your content can be shown in a fixed, simple user interface, do so.

To enable live resizing, you need to provide a resize control and an event handler for when it is clicked upon:

```
    <img id='resize' src='/System/Library/WidgetResources/resize.png'
onmousedown='mouseDown(event);'/>
```

Also, the resize control is placed in the bottom-right corner of the widget using CSS in your style sheet:

```
#resize {
    position:absolute;
    top: 208px;
    right: 2px;
    -apple-dashboard-region: dashboard-region(control rectangle);
}
```

In your JavaScript file, include this code:

```
var growboxInset;
```

```
function mouseDown(event)
{

    document.addEventListener("mousemove", mouseMove, true);
    document.addEventListener("mouseup", mouseUp, true);

    growboxInset = {x:(window.innerWidth - event.x), y:(window.innerHeight -
event.y)};

    event.stopPropagation();
    event.preventDefault();
}

function mouseMove(event)
{

    var x = event.x + growboxInset.x;
    var y = event.y + growboxInset.y;

    document.getElementById("resize").style.top = (y-12);
    window.resizeTo(x,y);

    event.stopPropagation();
    event.preventDefault();
}


function mouseUp(event)
{
    document.removeEventListener("mousemove", mouseMove, true);
    document.removeEventListener("mouseup", mouseUp, true);

    event.stopPropagation();
    event.preventDefault();
}
```

The three functions in this code handle the different mouse events that happen during a drag. First, `mouseDown` is called when the resize control is clicked upon. It records the initial placement of the click in `lastPos` and registers two handlers for the when the mouse moves and the mouse click ends.

Next, `mouseMove` is called every time the mouse is moved any distance. The code as listed here has no constraints, meaning that the widget can be as large or small as the user wants. If you have size constraints on your widget, add them here.

Finally, `mouseUp` is called when the mouse click ends. It removes itself and the `mouseMove` function as handlers. If you don't do this, these functions are still called whenever a mouse moves or a click ends.

Adjusting the Close Box

Depending on how your widget resizes and in which directions, you may need to adjust the placement of your widget's close box. The `setCloseBoxOffset` method gives you the ability to do this:

```
widget.setCloseBoxOffset(x,y);
```

The `x` and `y` coordinates that you provide are in relation to the top-left corner of the widget, where the values `0`, `0` places the center of the close box over the actual top-left corner of the widget window. The `x` and `y` values can not be larger than 100.

Using the Canvas

Safari, Dashboard, and WebKit-based applications support the JavaScript *canvas* object. The canvas allows you to easily draw arbitrary content within your HTML content.

Introduction to the Canvas

A canvas is an HTML tag that defines a custom drawing region within your web content. You can then access the canvas as a JavaScript object and draw upon it using features similar to Mac OS X's Quartz drawing system. The World Clock Dashboard widget (available on all Apple machines running Mac OS X version 10.4 or later) shows a good example , though using a canvas is by no means exclusive to Dashboard.

There are two steps to using a canvas in your web page: defining a content area, and drawing to the canvas object in the script section of your HTML.

Defining the Canvas

To use a canvas in your web page you first set up the drawing region. The World Clock Dashboard widget designates this region with the following code:

```
<canvas id="myCanvas" width='172' height='172'/>
```

In this context, the attributes of `<canvas>` worth noting are `id`, `width`, and `height`.

The `id` attribute is a custom identifier used to target a particular canvas object when drawing. The `width` and `height` attributes specify the size of the canvas region.

Within the World Clock widget, this area is defined to be the canvas:

Figure 1 The World Clock canvas region

Now that the canvas region has been defined, it is ready to be filled.

Drawing on a Canvas

Once you have defined the canvas area, you can write code to draw your content. Before you can do this, you need to obtain the canvas and its drawing context. The context handles the actual rendering of your content. The World Clock widget does this in its `drawHands()` function:

```
function drawHands (hoursAngle, minutesAngle, secondsAngle)
{
    var canvas = document.getElementById("myCanvas");
    var context = canvas.getContext("2d");
```

This function draws the hour, minute, and second hands on the face of the World Clock. As parameters, it takes the angles at which the three hands should be rotated as passed in by its caller.

You first query the JavaScript environment for the previously defined canvas, using its unique identifier: the `id` attribute in the `<canvas>` tag.

Once your script has acquired the canvas, you need to obtain its context. Using the `getContext("2d")` method, assign the canvas' draw context it to the `context` variable. From this point on, you call all operations intended for the canvas on `context`.

The first operation you perform empties the canvas. As the `drawHands()` function is called every second, it is important to empty it each time, so that the previously drawn configuration doesn't simply draw on top of the new configuration. The entire region, as defined by standard coordinates in the `<canvas>` tag, is cleared:

```
    context.clearRect(0, 0, 172, 172);
```

Next, you save the state of the original context space so that you can restore later. In the original context, the origin (the 0,0 coordinate) of the canvas is in the top left corner. Upon completion of the upcoming drawing code, you want to return to this context. Use the context's save method to do so:

```
    context.save();
```

Since you want the hands of the clock to rotate around the center of the clock, translate the origin of the context space to the center of the canvas:

```
    context.translate(172/2, 172/2);
```

Then draw the hour hand on the face of the clock. You copy the current context (with the origin at the center of the clock face), so that it can restored later. Then, you rotate the entire context, so that the y-axis aligns itself with the angle that the hour hand should point towards. Next, you draw the hour hand image (created in the code as a JavaScript `Image` object). The method `drawImage()` has five parameters: the image to be drawn, the x and y coordinate for the bottom left hand corner of the image, and the width and height of the image. Remember that while you draw the image as going straight up within the graphics context, you rotated the context to be at the correct angle for the hour hand:

```
    context.save();
    context.rotate(hoursAngle);
    context.drawImage(hourhand, -4, -28, 9, 25);
    context.restore();
```

Once you draw the hand, you restore the last saved context. This means that the context that you saved four lines prior, with its origin at the center of the canvas but not yet rotated, will be the active context again.

Use a similar procedure to draw the minute hand on the face of the clock. The differences this time are in the angle you rotate the context to and the size of the minute hand. Note that you save and rotate the context again, and then you restore it to its previous state, so that you can draw the next element independent of the rotation needed for the minute hand:

```
    context.save();
    context.rotate (minutesAngle);
    context.drawImage (minhand, -8, -44, 18, 53);
    context.restore();
```

Finally, draw the second hand. Note that this time, the context should not be saved and restored. Since this is the last time anything will be drawn in this particular context (with the origin at the center of the canvas), it is not necessary for you to save and restore again:

```
    context.rotate (secondsAngle);
    context.drawImage (sechand, -4, -52, 8, 57);
    context.restore();
}
```

Now that the clock face has been drawn, you should restore the context to its original state, as saved before any drawing occurred. This prepares the canvas for any future drawing that will occur, and gives you a consistent origin (the top-left corner of the canvas) to work from.

Remember, all of these techniques can be applied to a canvas object within any WebKit-based application. For more information on the canvas see `Canvas`.

Using the Pasteboard From JavaScript

Safari, Dashboard, and WebKit-based applications include support to let you handle cut, copy, and paste operations of your HTML content.

Introduction to JavaScript Pasteboard Operations

Support for pasteboard operations is implemented in JavaScript and may be applied to any element of your HTML page. To handle these operations, you provide functions to handle any of six JavaScript events:

- `onbeforecut`
- `oncut`
- `onbeforecopy`
- `oncopy`
- `onbeforepaste`
- `onpaste`

You can provide handlers for the `oncut`, `oncopy`, and `onpaste` events if you want to define custom behavior for the corresponding operations. You can also provide handlers for the `onbeforecut`, `onbeforecopy`, and `onbeforepaste` events if you want to manipulate the target data before it is actually cut, copied, or pasted.

If your `oncut`, `oncopy`, and `onpaste` handlers do the actual work of cutting, copying, or pasting the data, your handler must call the `preventDefault` method of the event object. This method takes no parameters and notifies WebKit that your handler takes care of moving the data to or from the pasteboard. If you do not call this method, WebKit takes responsibility for moving the data. You do not need to call `preventDefault` if you simply want to be notified when the events occur.

Adding Pasteboard Handlers to Elements

You can add handlers for pasteboard events to any element in a web page. When a pasteboard operation begins, WebKit looks for the appropriate handler on the element that is the focus of the operation. If that element does not define a handler, WebKit walks up the list of parent elements until it finds one that does. (If no element defines a handler, WebKit applies the default behavior.) To demonstrate this process, suppose you have the following basic HTML in a web page:

```
<body oncut="MyBodyCutFunction()"
        oncopy="MyBodyCopyFunction()"
        onpaste="MyBodyPasteFunction()">
    <span onpaste="MySpanPasteFunction()">Cut, copy, or paste here.</span>
</body>
```

If a user initiates a cut or copy operation on the text in the `span` tag, WebKit calls `MyBodyCutFunction` or `MyBodyCopyFunction` to handle the event. However, if the user tries to paste text into the span tag, WebKit calls the `MySpanPasteFunction` to handle the event. The `MyBodyPasteFunction` function would be called only if the paste operation occurred outside of the `span` tag.

Manipulating Pasteboard Data

When an event occurs, your handler uses the `clipboardData` object attached to the event to get and set the clipboard data. This object defines the `clearData`, `getData`, and `setData` methods to allow you to clear, get, and set the clipboard data.

> **Note:** For security purposes, the `getData` method can be called only from within the `onpaste` event handler.

WebKit's pasteboard implementation supports data types beyond those that are typically found in HTML documents. When you call either `getData` or `setData`, you specify the MIME type of the target data. For types it recognizes, including standard types found in HTML documents, WebKit maps the type to a known pasteboard type. However, you can also specify MIME types that correspond to any custom data formats your application understands. For most pasteboard operations, you will probably want to work with simple data types, such as plain text or a list of URIs.

WebKit also supports the ability to post the same data to the pasteboard in multiple formats. To add another format, you simply call `setData` once for each format, specifying the format's MIME type and a string of data that conforms to that type.

To get a list of types currently available on the pasteboard, you can use the `types` property of the `clipboardData` object. This property contains an array of strings with the MIME types of the available data.

Using Drag and Drop From JavaScript

Safari, Dashboard, and WebKit-based applications include support for customizing the behavior of drag and drop operations within your HTML pages.

Introduction to JavaScript Drag and Drop

Support for Drag and Drop operations is implemented in JavaScript and may be applied to individual elements of your HTML page. For drag operations, an element can handle the following JavaScript events:

- `ondragstart`
- `ondrag`
- `ondragend`

The `ondragstart` event initiates the drag operation. You can provide a handler for this event to initiate or cancel drag operations selectively. To cancel a drag operation, call the `cancelDefault` method of the event object. To handle an event, assign a value to the `effectAllowed` property and put the data for the drag in the `dataTransfer` object, which you can get from the event object. See "Changing Drag Effects" (page 93) for information on the `effectAllowed` property. See "Manipulating Dragged Data" (page 93) for information on handling the drag data.

Once a drag is under way, the `ondrag` event is fired continuously at the element to give it a chance to perform any tasks it wants to while the drag is in progress. Upon completion of the operation, the element receives the `ondragend` event and reports whether the drag was successful.

While a drag is in progress, events are sent to elements that are potential drop targets for the contents being dragged. Those elements can handle the following events:

- `ondragenter`
- `ondragover`
- `ondragleave`
- `ondrop`

The `ondragenter` and `ondragleave` events let the element know when the user's mouse enters or leaves the boundaries of the element. You can use these events to change the cursor or provide feedback as to whether a drop can occur on an element. The `ondragover` event is sent continuously while the mouse is over the element to give it a chance to perform any needed tasks. If the user releases the mouse button, the element receives an `ondrop` event, which gives it a chance to incorporate the dropped content.

If you implement handlers for the `ondragenter` and `ondragover` events, you should call the `preventDefault` method of the event object. This method takes no parameters and notifies WebKit that your handler will act as the receiver of any incoming data. If you do not call this method, WebKit receives the data and incorporates it for you. You do not need to call `preventDefault` if you simply want to be notified when the events occur.

Adding Handlers to Elements

You can add handlers for drag and drop events to any element in a web page. When a drag or drop operation occurs, WebKit looks for the appropriate handler on the element that is the focus of the operation. If that element does not define a handler, WebKit walks up the list of parent elements until it finds one that does. If no element defines a handler, WebKit applies the default behavior. To demonstrate this process, suppose you have the following basic HTML in a web page:

```
<body ondragstart="BodyDragHandler()"
        ondragend="BodyDragEndHandler()">
    <span ondragstart="SpanDragHandler()">Drag this text.</span>
</body>
```

If a user initiates a drag operation on the text in the `span` tag, WebKit calls `SpanDragHandler` to handle the event. When the drag operation finishes, WebKit calls the `BodyDragEndHandler` to handle the event.

Making an Element Draggable

WebKit provides automatic support to let users drag common items, such as images, links, and selected text. You can extend this support to include specific elements on an HTML page. For example, you could mark a particular `div` or `span` tag as draggable.

To mark an arbitrary element as draggable, add the `-khtml-user-drag` attribute to the style definition of the element. Because it is a cascading style sheet (CSS) attribute, you can include it as part of a style definition or as an inline style attribute on the element tag. The values for this attribute are listed in Table 1.

Table 1 Values for -khtml-user-drag attribute

Value	Description
`none`	Do not allow this element to be dragged.
`element`	Allow this element to be dragged.
`auto`	Use the default logic for determining whether the element should be dragged. (Images, links, and text selections can be dragged but all others cannot.) This is the default value.

The following example shows how you might use this attribute in a `span` tag to permit the dragging of the entire tag. When the user clicks on the `span` text, WebKit identifies the span as being draggable and initiates the drag operation.

```
<span style="color:rgb(22,255,22); -khtml-user-drag:element;">draggable
text</span>
```

Manipulating Dragged Data

When an event occurs, your handler uses the `dataTransfer` object attached to the event to get and set the clipboard data. This object defines the `clearData`, `getData`, and `setData` methods to allow you to clear, get, and set the data on the dragging pasteboard.

Note: For security purposes, the `getData` method can be called only from within the `ondrop` event handler.

Unlike many other browsers, the WebKit drag-and-drop implementation supports data types beyond those that are found in HTML documents. When you call either `getData` or `setData`, you specify the MIME type of the target data. For types it recognizes, WebKit maps the type to a known pasteboard type. However, you can also specify MIME types that correspond to any custom data formats your application understands. For most drag-and-drop operations, you will probably want to work with simple data types, such as plain text or a list of URIs.

Like applications, WebKit supports the ability to post the same data to the pasteboard in multiple formats. To add another format, you simply call the `setData` method with a different MIME type and a string of data that conforms to that type.

To get a list of types currently available on the pasteboard, you can use the `types` property of the `dataTransfer` object. This property contains an array of strings with the MIME types of the available data.

Changing Drag Effects

When dragging content from one place to another, it might not always make sense to move that content permanently to the destination. You might want to copy the data or create a link between the source and destination documents instead. To handle these situations, you can use the `effectAllowed` and `dropEffect` properties of the `dataTransfer` object to specify how you want data to be handled.

The `effectAllowed` property tells WebKit what types of operation the source element supports. You would typically set this property in your `ondragstart` event handler. The value for this property is a string, whose value can be one of those listed in Table 2.:

Table 2 Options for dragging and dropping an element

Value	Description
`none`	No drag operations are allowed on the element.
`copy`	The contents of the element should be copied to the destination only.
`link`	The contents of the element should be shared with the drop destination using a link back to the original.
`move`	The element should be moved to the destination only.
`copyLink`	The element can be copied or linked.
`copyMove`	The element can be copied or moved. This is the default value.

Value	Description
`linkMove`	The element can be linked or moved.
`all`	The element can be copied, moved, or linked.

The `dropEffect` property specifies the single operation supported by the drop target (`copy`, `move`, `link`, or `none`). When an element receives an `ondragenter` event, you should set the value of this property to one of those values, preferably one that is also listed in the `effectAllowed` property. If you do not specify a value for this property, WebKit chooses one based on the available operations (as specified in `effectAllowed`). Copy operations have priority over move operations, which have priority over link operations.

When these properties are set by the source and target elements, WebKit displays feedback to the user about what type of operation will occur if the dragged element is dropped. For example, if the dragged element supports all operations but the drop target only supports copy operations, WebKit displays feedback indicating a copy operation would occur.

Changing the Appearance of Dragged Elements

During a drag operation, WebKit provides feedback to the user by displaying an image of the dragged content under the mouse. The default image used by WebKit is a snapshot of the element being dragged, but you can change this image to suit your needs.

Changing the Snapshot With CSS

The simplest way to change the drag-image appearance is to use cascading style sheet entries for draggable elements. WebKit defines the `-khtml-drag` pseudoclass, which you can use to modify the style definitions for a particular class during a drag operation. To use this pseudoclass, create a new empty style-sheet class entry with the name of the class you want to modify, followed by a a colon and the string `-khtml-drag`. In the style definition of this new class, change or add attributes to specify the differences in appearance between the original element and the element while it is being dragged.

The following example shows the style-sheet definition for an element. During normal display, the appearance of the element is determined by the style-sheet definition of the `divSrc4` class. When the element is dragged, WebKit changes the background color to match the color specified in the `divSrc4:-khtml-drag` pseudoclass.

```
#divSrc4 {
    display:inline-block;
    margin:6;
    position:relative;
    top:20px;
    width:100px;
    height:50px;
    background-color:rgb(202,232,255);
}

#divSrc4:-khtml-drag {
    background-color:rgb(255,255,154)
}
```

Specifying a Custom Drag Image

Another way to change the drag image for an element is to specify a custom image. When a drag operation begins, you can use the `setDragImage` method of the `dataTransfer` object. This method has the following definition:

```
function setDragImage(image, x, y)
```

The `image` parameter can contain either a JavaScript Image object or another element. If you specify an Image object, WebKit uses that image as the drag image for the element. If you specify an element, WebKit takes a snapshot of the element you specify (including its child elements) and uses that snapshot as the drag image instead.

The `x` and `y` parameters of `setDragImage` specify the point of the image that should be placed directly under the mouse. This value is typically the location of the mouse click that initiated the drag, with respect to the upper-left corner of the element being manipulated.

Unfortunately, obtaining this information in a cross-browser fashion is easier said than done. There is no standard way to determine the position of the mouse relative to the document because different browsers implement the standard event values in subtly incompatible ways.

For the purposes of Safari and WebKit, `clientX` and `clientY` are document relative, as are `pageX` and `pageY` (which are thus always equal to `clientX` and `clientY`). For other browsers, Evolt.org has an article that describes how to obtain the mouse position in a cross-browser fashion, including sample code, at http://evolt.org/article/Mission_Impossible_mouse_position/17/23335/index.html.

Obtaining the position of the element under the mouse is somewhat easier. QuirksMode has a page (with code samples) on the subject at http://www.quirksmode.org/js/findpos.html.

Cross-Browser Compatibility

The drag-and-drop functionality built into WebKit and Safari works similarly to support in Microsoft Internet Explorer. The functionality built into FireFox and other Gecko-based browsers is very different, however.

In currently released versions of FireFox, this form of drag and drop is not generally supported from ordinary webpages (signed XUL applications notwithstanding) because you cannot register a drop target without loading an XPCOM component. Thus, with the exception of dropping things onto text areas (which are already drop targets), drag and drop must be emulated on this browser using mouse event handlers such as `onmouseup`.

As a result of differences in browser support, most web developers who need drag-and-drop support use libraries that mask browser differences. Some of these include the Dojo Tookit, DOM-Drag, ToolMan, Rico, and others.

Localizing Widgets

Localizing your widget provides a more comfortable and pleasant experience for foreign language speakers. If your widget is used in areas where languages other than English are spoken, you should localize it.

There are two sides to localizing your widget:

- What Dashboard does for you
- What you need to provide Dashboard

In addition to localizing your content, this chapter covers how to localize your widget's name in the Finder and the widget bar.

Note: The *Hello Welt* example provides sample code that shows how to localize a widget.

Language Projects

Before talking about localizing a Dashboard widget, you should be familiar with how Mac OS X handles localization. For most applications on Mac OS X, localized resources such as images, strings, and nib files exist within the application's bundle in `Contents/Resources/`. Each language gets its own directory, named after the language whose resources it holds. The names and location within the bundle are strict, as Mac OS X is expecting them to be there if a localization is requested. These folders are called *language project directories* and always end in the extension `.lproj`.

When an application is launched, the executable asks Mac OS X for certain localized resources. When this happens, Mac OS X looks for a language project within the application's bundle that corresponds with the first entry in the language precedence list, as set in System Preferences. If no language project for the preferred language is found, Mac OS X looks for a language project corresponding to the next language in the precedence list, and so on. Note that this process is mostly automatic, in that the application doesn't do any of the actual searching for language projects; it simply requests resources and Mac OS X provides them.

More on changing language and local preferences can be found in Language and Local Preferences.

What Dashboard Does for You

Widgets running within Dashboard use a similar process as Mac OS X applications when trying to load resources. Any time a resource load occurs in your code, Dashboard first looks for it within the language project directories in the Widget bundle. If Dashboard finds the resource within that language project directory, it provides it back to the widget. If not, searches through the rest of the language project directories, based on the precedence set in System Preferences. Finally, if the resource is not found in any language projects, Dashboard looks relative to the root level of the bundle.

Dashboard will look for localized resources in the following contexts:

- Any time the `@import` directive is used
- Any time the `src` attribute is used, including (but not limited to):
  - `<script src='myLogic.js' />`
  - `<img src="myImage.png">`
- Any other resource load targeted within your widget bundle

This is an additional reason behind recommending that you split your markup, logic, and design into separate files, as discussed in *Dashboard Programming Guide*.

What You Need to Provide Dashboard

When localizing your widget, provide Dashboard with localized versions of your resources. These include, but are not limited to, any strings that your widgets displays, images that change based on a language or region, and language-specific layouts. If you import a style sheet into your widget instead of including it in your HTML file, you'll be able to provide localized style sheets as well.

Each language you localize your widget into needs its own language project directory. In it you place all of the localized resources for that language. Each language project directory needs to be located at the root level of your widget. Table 16 (page 98) lists of common languages and their corresponding language project directory names:

Table 16 Common languages and corresponding language project names

Language	Language project directory name
Chinese (Simplified)	`zh-Hans.lproj`
Chinese (Traditional)	`zh-Hant.lproj`
Danish	`da.lproj`
Dutch	`nl.lproj`
English	`en.lproj`
Finnish	`fi.lproj`

Language	Language project directory name
French	`fr.lproj`
German	`de.lproj`
Italian	`it.lproj`
Japanese	`ja.lproj`
Korean	`ko.lproj`
Norwegian	`nb.lproj`
Portuguese	`pt.lproj`
Swedish	`sv.lproj`
Spanish	`es.lproj`

Note that these are just some of the possible localizations available. Mac OS X and Dashboard support many more languages and locals. Language Designations discusses language project directory naming conventions used for localizing applications on Mac OS X.

Localized Strings Example

An example for widget localization is to have all of your widget's strings localized. In each localized strings file, you provide an array of strings whose index is a variable common to all of the localized string files. You then include that file in your HTML file, and when you need a string, you simply retrieve it from the array.

The first step to implementing this scheme is to have a uniformly named file containing the strings inside of properly named language project directories. For example, having a file named `LocalizedStrings.js` inside each of your language project directories. The file looks like this for the German localization:

```
var localizedStrings = new Array;

localizedStrings['Hello, World!'] = 'Hallo, Welt!';
localizedStrings['Default'] = 'German';
```

Notice that the index into the `localizedStrings` array is a string. This is useful when combined with an accessor method that tries to retrieve the localized string:

```
function getLocalizedString (key)
{
    try {
        var ret = localizedStrings[key];
        if (ret === undefined)
            ret = key;
        return ret;
    } catch (ex) {}

    return key;
}
```

The advantages to this are twofold: first, the index for a string is memorable, and secondly, if the string retrieval fails, the key string is returned. This way, you are assured that some string will always be returned, no matter the circumstances. This is especially valuable when you are testing your widget in Safari.

Finally, you'll need to use the localized string in your widget. This code ties together all of these previous concepts and inserts the string into your widget:

```
function setup()
{

    document.getElementById('helloText').innerText = getLocalizedString('Hello,
 World!');
   document.getElementById('language').innerText = getLocalizedString('Default');
}
```

Since the proper localized strings file is already loaded, this will fetch the localized equivalent of "Hello, World!" in the `localizedStrings` array and placing it in your layout.

Note: The "HelloWelt" sample code also includes localized style sheets in each language project directory. This allows the design of the widget to vary based on the language. Remember to take varying string lengths into account when localizing your widget.

Localized Widget Names

In addition to localizing the content of your widget, you should localize your widget's name. The name is displayed in the Finder and the widget bar and is pulled from your `Info.plist` information property list file and localized `InfoPlist.strings` files.

In your Info.plist, you need to specify the key `CFBundleDisplayName` and provide a corresponding value:

```
<key>CFBundleDisplayName</key>
<string>Hello World</string>
```

This value is a default value that's used if no localized string can be found. It also needs to be the name of your widget on disk, without the `.wdgt` file extension. Inside of each language project directory in your widget, place a file named `InfoPlist.strings` and in it provide the proper localized name using this format:

```
CFBundleDisplayName = "Hallo Welt";
```

For a more in-depth look at using `CFBundleDisplayName`, read *Runtime Configuration Guidelines*.

Specifying Access Keys

If your widget needs resources that extend beyond your widget's bundle or HTML, CSS, and JavaScript technologies, you need to take Dashboard's Info.plist Access keys into account.

Using Access Keys

Dashboard allows you to "declare your intentions" when you:

- Access files outside of your widget bundle
- Use a WebKit or standard browser plug-in
- Access network resources
- Run a Java applet
- Run a command-line utility
- Use a widget plug-in

"Declaring your intentions" means that before your widget is run, you specify in your widget's information property list file which resources you want to use. The keys and their meaning are listed in Table 17 (page 101):

Table 17 Info.plist Keys for the Widget resource access

Key	Type	Definition	Example
`AllowFileAccess-OutsideOfWidget`	Boolean	Access to files across the file system; limited by the user's permissions.	`<img src="~/Sites/images/macosxlogo.gif">`
`AllowFullAccess`	Boolean	Access to the file system, WebKit and standard browser plug-ins, Java applets, network resources, and command-line utilities.	N/A

Key	Type	Definition	Example
`AllowInternetPlugins`	Boolean	Access to WebKit and standard browser plug-ins, such as QuickTime.	`<embed src="http://www.foo.com/bar.mov" type="video/quicktime" width="320" height="256"></embed>`
`AllowJava`	Boolean	Access to Java applets.	`<applet code="foo.class" width="320" height="256"></applet>`
`AllowNetworkAccess`	Boolean	Access to any resources that are not file-based, including those acquired through the network.	`<img src="http://www.foo.com/bar.png">`
`AllowSystem`	Boolean	Access to command-line utilities using the widget script object.	`var s = widget.system("/usr/bin/foo", null);`
`Plugin`	String	Specifies a widget plug-in.	`foo.widgetplugin`

If you attempt to use any of these resources without first specifying them in your widget's information property list file, your attempt fails.

Accessing External Resources

Widgets can open applications and web pages outside of their bundle. If your widget provides a subset of information found on the Internet, a link to the full data set that opens in Safari is appropriate. If your widget interfaces with an application, for example, iTunes, it should open it first. Dashboard can do this all for you.

Note: Before reading this chapter, read "Specifying Access Keys" (page 101) to learn more about the widget access keys.

URL Opening

Sometimes you may want your widget to open a webpage when certain information is clicked. For instance, clicking a stock symbol in a stock ticker widget would probably load a webpage in the default browser displaying information relevant to the stock.

To open a webpage, use the `widget.openURL(url)` method. For example, you may use it inside of a function to dynamically assemble a URL using the contents of a variable you set elsewhere in your code:

Listing 7 Assembling a URL and passing it to `widget.openURL`

```
<html>
<head>
<script>
    ...
    function clicked(section)
    {
        if (widget)
        {
            widget.openURL('http://www.apple.com/' + section);
        }
    }
    ...
</script>
</head>
<body>
    ...
    <span onclick="clicked('developer/')">Developer</span>
    <span onclick="clicked('store/')">Store</span>
    ...
</body>
</html>
```

In Listing 7, an arbitrary function is called when a user clicks within some text. A portion of a URL is passed to the `clicked` function and then appended onto another string, which is then passed to the `openURL` method. It then opens the user's default browser with the provided URL.

Alternatively, you can embed the method in any `<span>` tag:

```
<span onclick="widget.openURL('http://www.apple.com/')">Apple</span>
```

Note: You must include `http://` in the `url` argument of the `widget.openURL` method or the URL will not open. If, for example, you passed in `'www.apple.com'`, instead of `'http://www.apple.com'`, the URL would not open.

Application Activation

In addition to being able to open a webpage, your widget can open applications. Calling `widget.openApplication()` dismisses Dashboard and either opens the specified application or, if it was already open, brings it to the forefront.

The parameter passed into this method is the bundle ID for an application. For instance, to open iTunes, you pass in the string `com.apple.iTunes`:

```
widget.openApplication("com.apple.iTunes");
```

Note that there is no facility for passing arguments to an application. For this level of interactivity between a widget and application, you could try one of these options:

- Use the `widget.system()` method, as discussed in "Accessing Command Line Utilities" (page 105), with the `open` command-line utility
- Implement a widget plug-in, as discussed in "Creating a Widget Plug-in" (page 111)

Accessing Command Line Utilities

Dashboard provides you with a method for using command-line utilities and scripts within your widget. With this capability you can use any standard utilities included with the system or any utilities or scripts you include within your widget.

Note: Before reading this chapter, read "Specifying Access Keys" (page 101) to learn more about the widget access keys.

The System Method

Running a command-line utility or script within your widget requires you to use the `widget.system()` method. The method is defined as:

```
widget.system("command", handler)
```

The parameters of the `widget.system()` method are:

Table 18 `widget.system()` parameters

Parameter	Definition	Example
`command`	A string that specifies a command-line utility; may contain parameters and flags.	`"/bin/ls -l -a"`
`handler`	A function called when the command-line utility finishes execution; toggles execution of the command between synchronous and asynchronous modes. If specified, the handler needs to accept an argument.	`systemHandler`

Note: When specifying the command, always include the full path to the command or the path to the command relative to the root level of the widget. If you are unsure what the path is, the command `which` can tell it to you.

Also, don't rely on non-standard enviroment variables, custom search paths, and Terminal preferences when passing a command to `widget.system()`.

Depending on what you pass into the handler parameter, your call to `widget.system()` will operate in one of two modes: synchronous or asynchronous.

Synchronous Operation

Using `widget.system()` synchronously means that you are going to hold up the execution of your widget until you get the results of the command you are running. You want to use them this way when working with commands that provide output once and execute in a short period of time.

Note: Using `widget.system()` synchronously is recommended only for development and debugging purposes. Do not deliver a widget that uses `widget.system()` synchronously; instead, use `widget.system()` asynchronously, discussed in "Asynchronous Operation" (page 106), when delivering a shipping widget.

An example of this would be if you wanted to run the command `id` from within your widget:

```
widget.system("/usr/bin/id -un", null);
```

The first argument specifies the command you want to run; here, you're running `id` with the flag `-un`. You have not specified an event handler for this command, so all execution in your widget halts until this command is finished.

Running `id` as shown above executes the command, but any output is lost since you don't specify that you want that information. To get its output, specify the `outputString` property and save it in a variable:

```
var output = widget.system("/usr/bin/id -un", null).outputString;
```

You can get either the output string, the error string, or the command's output status when using `widget.system()` synchronously:

Table 19 `widget.system()` properties during synchronous usage

Property	Definition	Usage
`outputString`	The output of the command, as placed on `stdout`.	`var output = widget.system("id -un", null).outputString;`
`errorString`	The output of the command, as placed on `stderr`.	`var error = widget.system("id -un", null).errorString;`
`status`	The exit status of the command.	`var status = widget.system("id -un", null).status;`

Asynchronous Operation

Providing a handler as the second argument of `widget.system()` runs the command in asynchronous mode. This means that execution within your widget continues while the command is executing. The handler that you specify is called when the command is finished and needs to accept a single object as a parameter. That object contains the last output of the command as it finishes execution. You can retrieve these properties from it:

Table 20 `widget.system()` end handler parameter object properties

Property	Definition
`object.outputString`	The last output of the command, as placed on `stdout`.
object.`errorString`	The last output of the command, as placed on `stderr`.
`object.status`	The exit status of the command.

Because the command is running asynchronously, it may be necessary to interact with the command during its execution. Using `widget.system()` asynchronously returns an object that you can use for further interaction with the command:

```
var myCommand = widget.system("/sbin/ping foo.bar", endHandler);
```

The object returned (and saved in `myCommand`) responds to a number of methods and has various properties:

Table 21 `widget.system()` properties and methods available during asynchronous usage

Option	Purpose	Description
`myCommand.outputString`	Property	The current string written to `stdout` (standard output) by the command.
`myCommand.errorString`	Property	The current string written to `stderr` (standard error output) by the command.
`myCommand.status`	Property	The command's exit status, as defined by the command.
`myCommand.onreadoutput`	Event Handler	A function called whenever the command writes to `stdout`. The handler must accept a single argument; when called, the argument contains the current string placed on `stdout`.
`myCommand.onreaderror`	Event Handler	A function called whenever the command writes to `stderr`. The handler must accept a single argument; when called, the argument contains the current string placed on `stderr`.
`myCommand.cancel()`	Method	Cancels the execution of the command.
`myCommand.write(string)`	Method	Writes a string to `stdin` (standard input).
`myCommand.close()`	Method	Closes `stdin` (`EOF`).

For instance, to run the command `ping` and be notified every time it writes something to `stdout`, use this code:

```
var myCommand = widget.system("/sbin/ping foo.bar", endHandler);
myCommand.onreadoutput = outputHandler;
```

Alternatively, you can use:

```
widget.system("/sbin/ping foo.bar", endHandler).onreadoutput = outputHandler;
```

Your `onreadoutput` handler should accept an argument. When it is called, it is passed a string that has the most recent string placed on `stdout`:

```
function outputHandler(currentStringOnStdout){    // Code that does something
with the command's current output like...
    document.getElementById("element").innerText = currentStringOnStdout;}
```

Commands such as `ping` run indefinitely, so you probably want to end its execution at some point. Use the `cancel()` method on the object that you receive from `widget.system()` to do this:

```
myCommand.cancel();
```

Other commands, such as `bc`, require input at some point in their execution. To write to standard input (where these commands expect their input), use the `write()` method:

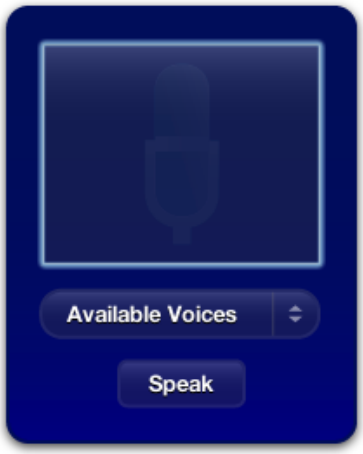
```
myCommand.write("8*5");
```

To close these commands properly (using the end-of-file, or `EOF`, signal), use `close()`:

```
myCommand.close();
```

Don't forget that in order for this command to run asynchronously, you need to provide an event handler for the end of execution. This handler is passed the same object that is created when you first use `widget.system()`. That means that you can get the command's status code or, if you didn't use the `onreadoutput` or `onreaderror` handlers, you can obtain the command's complete output to `stdout` or `stderr`, respectively.

Sample Code



The Voices sample widget (available in the *Voices* sample project) uses `widget.system()` asynchronously. When Voices is first opened, it introduces itself:

```
function setup()
{
    if(window.widget) {
```

```
        currentlyBeingSpoken = widget.system("/usr/bin/osascript -e 'say \"Welcome
 to Voices!\" using \"Fred\"'" , doneSpeaking);
    }
}
```

By specifying a handler for when the command is finished executing, the command runs asynchronously. A global variable, `currentlyBeingSpoken`, is assigned the command object so that commands can be issued to it during its execution, if needed. The `doneSpeaking()` function, called when the command is done, sets `currentlyBeingSpoken` to `NULL`.

Later, when a user inputs a phrase to be spoken, this code is called:

```
if(window.widget) {
    if(currentlyBeingSpoken != null) {
        currentlyBeingSpoken.cancel();
    }
    currentlyBeingSpoken = widget.system("/usr/bin/osascript -e 'say \"" +
textToSpeak + "\" using \"" + chosenVoice + "\"'" , done);
}
```

Here `currentlyBeingSpoken` is checked to see if a command is already in execution. If so, the `cancel()` method is called on it to stop it and then a new command is issued. The `done()` function performs some user interface housekeeping and then calls `doneSpeaking()` to set `currentlyBeingSpoken` to `NULL`.

Voices also has each voice introduce itself when it is selected in a menu. This code follows a similar logic as the previous sample:

```
function voiceChanged(elem)
{
    var chosenVoice = elem.options[elem.selectedIndex].value;
    document.getElementById("voiceMenuText").innerText = chosenVoice;

    if(window.widget) {
        if(currentlyBeingSpoken != null) {
            currentlyBeingSpoken.cancel();
            done();
        }
        currentlyBeingSpoken = widget.system(
                    "/usr/bin/osascript -e 'say \"Hi, I`m " +
                    chosenVoice + ".\" using \"" +
                    chosenVoice + "\"'" ,
                    doneSpeaking
                    );
    }
}
```

Creating a Widget Plug-in

Widgets alone cannot access applications directly, receive distributed notifications, or read files from disk. To enable these interactions, you need to provide a plug-in. You are required to implement an interface for your plug-in that makes itself available to the widget. This interface communicates with your application in whatever manner is most appropriate, for example, by issuing AppleScript commands.

You can use a widget as another way to provide an interface to an application. Providing a widget front end allows a user to interact with your application in an unobtrusive and simple way that is easily accessible.

A widget plug-in is a Cocoa bundle. In Xcode, use the "Cocoa Bundle" template to create a bundle. In the plug-in code, implement the widget plug-in interface.

For examples of widgets that use plug-ins, see *Birthdays* and *Reminders* sample code projects.

Note: Before reading this chapter, read "Specifying Access Keys" (page 101) to learn more about the widget access keys.

Widget Plug-in Interface

Any widget plug-in must implement this method in order to be used from within Dashboard:

```
- (id) initWithWebView:(WebView*)webview
```

Dashboard calls this when your plug-in is first loaded. At this point, initialize your principal class and prepare any critical data structures.

To have your plug-in interact with your widget, you will need to implement the WebScripting interface, as defined in Using Objective-C From JavaScript (page 115) and `WebScripting`. In addition to this interface, you also need to implement this method:

```
- (void) windowScriptObjectAvailable:(WebScriptObject *)windowScriptObject
```

If implemented, Dashboard calls it before your widget is loaded and allows you to add JavaScript objects that your widget can use. These objects bridge the gap between JavaScript and Objective-C, and are your interface with your widget. After this message is received, call `setValue: forKey:` on the just-received `WebScriptObject` to bind it to your own object and to give it a name. In order to function properly, the object that you bind to the `WebScriptObject` must implement the WebScripting interface.

This example demonstrates what your implementation of this method should include:

```
- (void) windowScriptObjectAvailable:(WebScriptObject *) windowScriptObject
{
    [windowScriptObject setValue:self forKey:@"MyWindowScriptObject"];
    ...
}
```

Any methods that belong to the object that you bind to the given `windowScriptObject` will be available to your widget in JavaScript, via the specified key. However, your methods will be available using the default name created for it, which can be confusing depending on its Objective-C name. Developers are advised to implement this method to provide a more human-readable name:

```
+ (NSString *)webScriptNameForSelector:(SEL)aSelector
```

In the following example, your plug-in class is bound to a received `WebScriptObject`, named `windowScriptObject`. The key for the object is `MyWindowScriptObject`, meaning that, from within the widget, any method belonging to the `MyWindowScriptObject` class may be called upon it:

```
<html>
<head>
...
<script>
...
function someFunction()
{
    ...
    if (MyWindowScriptObject)
    {
        MyWindowScriptObject.someMethod(someArg);
    }
    ...
}
...
</script>
</head>
...
</html>
```

For example, you can use this to notify the plug-in when the widget is finished loading in Dashboard. You can set up a function to be called when the widget has finished loading. This function will, in turn, call any method you supply:

```
<html>
...
<body onload='MyWindowScriptObject.someMethod(someArg)'>
...
</body>
</html>
```

Widget Plug-in Bundle

The Xcode standard information property list file provides most of the information you need for the plug-in to function properly. Despite this, you must provide a value for the `NSPrincipalClass` property.

Once you compile the bundle, you are ready to deploy it. For your widget to use your plugin, place it at the root level of your widget bundle.

In order for your plug-in to be loaded when you activate your widget, it needs to be specified in your widget's `Info.plist` file. The property `Plugin` needs to be added, and its value should be a String filled with the name of your bundle.

Additional Resources

For more information on Dashboard plug-ins, see *Dashboard Reference* in Apple Applications Documentation.

To learn more about bridging your widget's JavaScript environment with your widget plug-in's Cocoa bundle, read Using Objective-C From JavaScript (page 115).

When compiling your widget plug-in, make sure you're building it as a Universal plug-in for use on PowerPC and Intel-based Macintosh computers. For more on Universal binaries, read Technical Q&A QA1451: Intel-Based Macs, Dashboard, Safari, and You and *Universal Binary Programming Guidelines, Second Edition*.

Using Objective-C From JavaScript

The web scripting capabilities of WebKit permit you to access Objective-C properties and call Objective-C methods from the JavaScript scripting environment.

An important but not necessarily obvious fact about this bridge is that it does *not* allow *any* JavaScript script to access Objective-C. You cannot access Objective-C properties and methods from a web browser unless a custom plug-in has been installed. The bridge is intended for people using custom plug-ins and JavaScript environments enclosed within WebKit objects (for example, a WebView).

How to Use Objective-C in JavaScript

The WebScripting informal protocol, defined in `WebScriptObject.h`, defines methods that you can implement in your Objective-C classes to expose their interfaces to a scripting environment such as JavaScript. Methods and properties can both be exposed. To make a method valid for export, you must assure that its return type and all its arguments are Objective-C objects or basic data types like `int` and `float`. Structures and non object pointers will not be passed to JavaScript.

Method argument and return types are converted to appropriate types for the scripting environment. For example:

- JavaScript numbers are converted to NSNumber objects or basic data types like `int` and `float`.
- JavaScript strings are converted to NSString objects.
- JavaScript arrays are mapped to NSArray objects.
- Other JavaScript objects are wrapped as WebScriptObject instances.

Instances of all other classes are wrapped before being passed to the script, and unwrapped as they return to Objective-C.

A Sample Objective-C Class

Let's look at a sample class. In this case, we will create an Objective-C address book class and expose it to JavaScript. Let's start with the class definition:

```
@interface BasicAddressBook: NSObject {
}
+ (BasicAddressBook *)addressBook;
- (NSString *)nameAtIndex:(int)index;
@end
```

Now we'll write the code to publish a `BasicAddressBook` instance to JavaScript:

```
BasicAddressBook *littleBlackBook = [BasicAddressBook addressBook];

id win = [webView windowScriptObject];
[win setValue:littleBlackBook forKey:@"AddressBook"];
```

That's all it takes. You can now access your basic address book from the JavaScript environment and perform actions on it using standard JavaScript functions. Let's make an example showing how you can use the `BasicAddressBook` class instance in JavaScript. In this case, we'll print the name of a person at a certain index in our address book:

```
function printNameAtIndex(index) {
    var myaddressbook = window.AddressBook;
    var name = myaddressbook.nameAtIndex_(index);
    document.write(name);
}
```

You may have noticed one oddity in the previous code example. There is an underscore after the JavaScript call to the Objective-C `nameAtIndex` method. In JavaScript, it is called `nameAtIndex_`. This is an example of the default method renaming scheme in action.

Unless you implement `webScriptNameForSelector` to return a custom name, the default construction scheme will be used. It is your responsibility to ensure that the returned name is unique to the script invoking this method. If your implementation of `webScriptNameForSelector` returns `nil` or you do not implement it, the default name for the selector will be constructed as follows:

- Any colon (":") in the Objective-C selector is replaced by an underscore ("_").
- Any underscore in the Objective-C selector is prefixed with a dollar sign ("$").
- Any dollar sign in the Objective-C selector is prefixed with another dollar sign.

The following table shows example results of the default method name constructor:

Objective-C selector	Default script name for selector
setFlag:	setFlag_
setFlag:forKey:withAttributes:	setFlag_forKey_withAttributes_
propertiesForExample_Object:	propertiesForExample$_Object_
set_$:forKey:withDictionary:	set$_$$_forKey_withDictionary_

Since the default construction for a method name can be confusing depending on its Objective-C name, you would benefit yourself and the users of your class if you implement `webScriptNameForSelector` and return more human-readable names for your methods.

Getting back to the BasicAddressBook, now we'll implement `webScriptNameForSelector` for our `nameAtIndex` method. In our BasicAddressBook class implementation, we'll add this:

```
+ (NSString *) webScriptNameForSelector:(SEL)sel
{
    ...

    if (sel == @selector(nameAtIndex:))
            name = @"nameAtIndex";
```

```
    return name;
}
```

Now we can change our JavaScript code to reflect our more logical method name:

```
function printNameAtIndex(index) {
    var myaddressbook = window.AddressBook;
    var name = myaddressbook.nameAtIndex(index);
    document.write(name);
}
```

For security reasons, no methods or KVC keys are exposed to the JavaScript environment by default. Instead a class must implement these methods:

```
+ (BOOL)isSelectorExcludedFromWebScript:(SEL)aSelector;
+ (BOOL)isKeyExcludedFromWebScript:(const char *)name;
```

The default is to exclude all selectors and keys. Returning NO for some selectors and key names will expose those selectors or keys to JavaScript.

See *WebKit Objective-C Framework Reference* for all the information on excluding methods and properties from the JavaScript environment.

Delivering Widgets

After you've created your widget, you need to distribute it to your customers. This chapter outlines steps that you should take to ensure that your customers have a pleasant experience downloading and installing your widget.

Packaging Your Widget

Widgets are much less complex than applications and should provide a light-weight install experience. The preferred packaging experience is to have widgets delivered in zip archive format and placed on your web server for download. Only archive the `.wdgt` bundle, omitting all other files. Link to the widget archive from your website to enable the download.

When downloaded using Safari, the zipped widget is automatically unarchived and the user is presented with an install dialog, asking them if they want the widget to be installed. When downloaded using other web browsers, users need to manually open the widget to show the widget installer.

Delivery Tips

Here are some tips for you to keep in mind when readying your widget for delivery:

- Follow these steps to create a zip archive:
  - Select the widget in the Finder
  - Choose File > Create Archive
  - Upload resulting archived widget to your web server
- Avoid multi-step installations, registration, and purchasing after the widget is downloaded. If registration, purchase, and the display of an End User License Agreement is required, perform these functions locally on your website prior to download. If it's necessary to communicate with your widget after download, use cookies.
- Include the instructions below on your widget download page for users to follow:

  Mac OS X v.10.4 Tiger is required. If you're using Safari, click the download link. When the widget download is complete, the widget installer appears. Click Install if you want the widget installed on your Mac. If you're using a browser other than Safari, click the download link. When the widget download is complete, unarchive and open it to show the widget installer.

Document Revision History

This table describes the changes to *Dashboard Programming Topics*.

Date	Notes
2009-02-04	Made minor corrections.
2009-01-06	Added an article that introduces Dashboard and widgets and describes how to create a simple widget.
2008-10-15	Made minor corrections.
2007-04-13	Added an article about the Dashboard Sync feature in Mac OS X v.10.5.
2006-08-07	Added information about using widget.system synchronously. Clarified the descriptions of some Apple Classes.
2006-04-04	Fixed typos in Apple Scroll Area and Apple Glass Button descriptions.
2006-01-10	Clarified directions on using Apple Classes with widget backs and preferences. Added links to Webscripting reference documentation. Fixed typos.
2005-12-06	Added information on JavaScript Apple classes for Dashboard. Changed the title from "Dashboard Programming Guide."