Dashboard Programming Guide

Tiger

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

Simultaneously published in the United States and Canada.

Contents

Tables and Figures

Introduction to Dashboard Programming Guide

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

Who Should Read This Document?

Dashboard Programming Guide is for anyone who wants to create a Dashboard widget. It will provide you with an understanding of the structure and basic requirements for a widget. You will also learn additional techniques you can use to improve your widget's functionality.

Organization of This Document

This document contains the following chapters:

- "Dashboard Overview" (page 11) talks about what Dashboard is and what makes a widget.
- "Widget Basics" (page 15) walks you through the creation of a sample widget. It discusses the internal structure of a widget and the files needed to make a widget work.
- "Design Conventions" (page 21) provides guidelines and tips for designing successful widgets.
- "Reverse Sides and Preferences" (page 33) tells you how to display, save, and retrieve preferences.
- "Events" (page 39) discusses Dashboard and widget events that your widget may want to be aware of.
- "Control Regions" (page 43) defines and explains how to work with control regions, areas where controls are present in a widget.
- "Resizing" (page 47) provides code useful for implementing resizing in your widget.
- "Localization" (page 51) discusses offering your widget with international users in mind, using localizable strings and other resources.
- "Security" (page 57) describes the widget security model, used to protect users from unsafe widgets.

- "External Access" (page 59) talks about opening applications or web pages in a browser with your widget.
- "Command-Line Access" (page 61) tells you how to access command-line utilities and scripts from within your widget.
- "Widget Plug-in" (page 67) discusses native code plug-ins that your widget uses to interact with other applications.
- "Widget Delivery" (page 71) 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*.

Web Kit, the technology that underlies Dashboard, offers additional features that you may find useful:

- The Canvas allows you to declare an arbitrary drawing context within your widget. With it you can draw as if you were using Quartz-like calls within a native application. Using the Canvas discusses this in depth.
- Supporting drag and drop may make your widget more useful, allowing users to drag files on to your widget for it to work with. Using Drag and Drop from JavaScript covers this concept.
- Implementing Pasteboard operations such as copy and paste may be appropriate for your widget. Using the Pasteboard with JavaScript shows how to implement this.
- Manipulating the Document Object Model may prove useful when coding your widget. Using the JavaScript Document Object Model explains the DOM and goes more in depth on using it from JavaScript.

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

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

Dashboard Overview

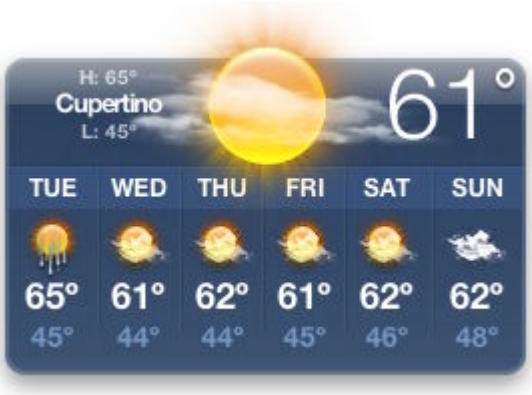
Mac OS X version 10.4 “Tiger” includes a new feature called Dashboard. Dashboard is a way to keep vital information at your fingertips, ready when you need it and easily hidden when you’re done with it. That information is presented in the form of widgets–miniature applications that live exclusively in Dashboard.

The Dashboard Environment

You show Dashboard by using a key stroke, as specified in the Exposé & Dashboard pane of System Preferences. By default, the key is F12. Alternatively, you can click on the Dashboard icon in the Dock. It overlays your current window set with Dashboard, the place where widgets live.

Dashboard itself is the environment where information and utilities are shown. The information and utilities are embodied in widgets. Multiple widgets can exist in Dashboard at any given time. Users have complete control over what widgets are visible and can freely move them anywhere they please within Dashboard. The widgets are shown and hidden along with Dashboard, and they share Dashboard. When Dashboard is dismissed, the widgets disappear along with it.

Dashboard Widgets



A widget is a mini-application that exists exclusively in Dashboard. From a user perspective, it behaves as a program should: it shows useful information or helps them obtain information with a minimum of required input.

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.

Widget or Application?

Before creating a widget, you need to decide what its functionality should be. Be careful to avoid having your widget do everything a full application would do.

An example of judicious use of a widget is to provide a front end for a time card application. The application provides all of the features needed by the user, while a companion widget lets you clock in and out and choose the job that you're currently working on.

The widgets that Apple provides with Mac OS X v10.4 can give you a clue as to the scope of a widget's functionality. For instance, the Stickies widget lets you type, copy, cut, and paste, but you can't save the contents of the widget. The Weather widget shows you, by default, the temperature and location,

along with a visual indicator of the current weather condition. Upon clicking the widget, a forecast is shown. Notice that the widget shows only one location; if users want to track more locations, they can simply add another instance of the Weather widget to their Dashboard.

As a rule, avoid making a widget that your users live in, meaning that they spend considerable time working in it to get serious tasks done. Widgets should provide information with little or no input or should perform simple tasks that a user may want to do often. Also, be respectful of the limited space available on Dashboard. If your widget is needlessly large, don't expect users to keep it around.

Widget Basics

The power of Dashboard comes in the form of its widgets. Widgets are the objects that users interact with when Dashboard is invoked. They perform tasks such as providing a clock or a calculator for the user. To develop a widget you must work with the bundle structure, a property list, and some combination of HTML, CSS, and JavaScript.

In this chapter, you create a simple "Hello, World!" widget:

In doing so, you take a look at a widget's bundle structure, its `Info.plist` property list, a basic style sheet, and the HTML file needed to make the widget function. Finally, you tie it all together and install the widget in the right place.

Note: The sample code included in this chapter 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. In the `HelloWorld` example widget, there is a fifth file that contains the style sheet for the widget.

For the HelloWorld widget, the bundle structure contains the following files:

```
HelloWorld.wdgt/
    Icon.png
    Info.plist
    Default.png
    HelloWorld.html
```

```
HelloWorld.css
```

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. Useu JavaScript to introduce interactivity into your widget.

While 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 2-1 (page 16). Splitting your markup, design, and logic into separate files may also make localization easier, as discussed later on in "Localization" (page 51). In general, file extensions are used to reflect the purpose of the file:

Table 2-1 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.

To load your CSS and JavaScript, you'll need to import them inside of your HTML file. For importing a style sheet, the markup looks like 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 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 widget's information property list file are eight mandatory values and one optional value. The properties are listed in Table 2-1, along with a definition and the value used in the sample `HelloWorld` widget:

Table 2-2 Widget `Info.plist` values

Property	Example	Definition
`CFBundleIdentifier`	`com.apple.widget.HelloWorld`	Required. A string that uniquely identifies the widget, in reverse domain format.
`CFBundleName`	`HelloWorld`	Required. A string that provides the name of your widget. Must match.
`CFBundleDisplayName`	`Hello World`	Required. A string that reflects 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`	`15`	Optional. An integer that sets the placement of the widget's close box on the x-axis.
`CloseBoxInsetY`	`15`	Optional. An integer 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 main 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 `CloseBoxInsetX` and `CloseBoxInsetY` values. These values determine the placement of the close box over the top-left corner of your widget. You should position the close box so that the "X" is centered over the top-left corner of the widget.

Icons and Default Images

The two PNG files required in a widget are the icon and default image files. The icon file, `Icon.png`, is used in the widget bar as a representation of your widget. The default image, `Default.png`, is shown in your while it 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. These files are required and Dashboard expects them to be named as outlined above.

Figure 2-1 The HelloWorld widget default image

Widget Implementation

Your widget's HTML file provides the implementation of the widget. It can be named anything but must reside at the root level of the widget bundle and must be specified in the `Info.plist` file. For the `HelloWorld` sample widget, the HTML file displays an image and the words "Hello, World!" Here are the contents of the `HelloWorld.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.You'll 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), it isn't required that you 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:

```
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 2-2 (page 19) shows what the code looks like when rendered in Safari.

Figure 2-2 The HelloWorld widget being previewed in Safari

In fact, when testing your widget, you can load it into Safari and get the same behavior that you would if it were loaded into Dashboard. The exception to this rule are interactions that rely specifically on the presence of Dashboard, as discussed in coming chapters.

Assembling the Widget

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

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

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

Once the files are in place, rename the directory `HelloWorld.wdgt`.

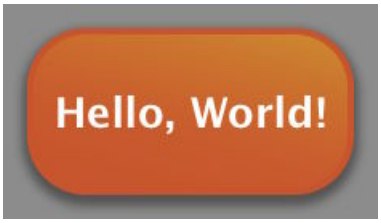
Note: If you do not have the "Show all file extensions" preference checked in the Finder, you are asked to confirm this action. Go ahead and choose Add and the bundle will be made for you. If you have this preference checked, the bundle is made without any prompting.

Once the bundle has been renamed, to install it, place it in one of these directories:

- `/Library/Widgets/`
- `~/Library/Widgets/`

Once installed, double-click your `HelloWorld.wdgt` widget. Dashboard is loaded and the widget appears. It should look like the view in Figure 2-3.

Figure 2-3 The HelloWorld widget installed and running in Dashboard



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

The power of widgets comes from their ability to use JavaScript and CSS to apply widget-specific properties to elements. In the remaining chapters, you add additional features to your widget to introduce interactivity, conform to expected behaviors, and expand interactions with other applications.

Design Conventions

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* 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 reverse). 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 3-1 A cluttered widget is a jack of all trades, master of none

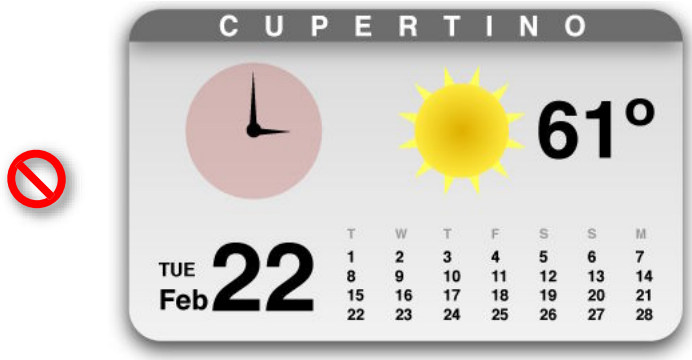


Figure 3-2 Three simple widgets, each focused on a single task

- 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 3-3 A large widget monopolizes valuable screen space

Figure 3-4 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 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 3-5 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 3-6 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.
- Avoid using Aqua controls on your main interface. Aqua controls should only be used for the reverse 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.

Figure 3-7 Aqua controls don't belong on the face of your widget

Figure 3-8 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 reverse side of the widget for information that isn't vital to the widget's purpose, such as branding, licensing information, and copyright notices.

Figure 3-9 Don't waste valuable space in your widget with advertising

Figure 3-10 Put information not vital to the widget on the reverse side

- 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
  - Buttons
  - Resize control

Widget Reverse Side Design Guidelines

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

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

- Use the info button graphic to signify that you are using the reverse side 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 3-11 A non-standard control for showing your widget's reverse

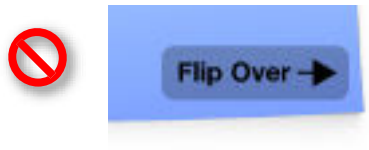
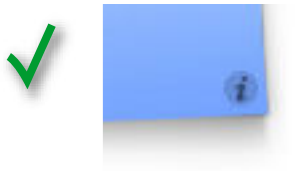
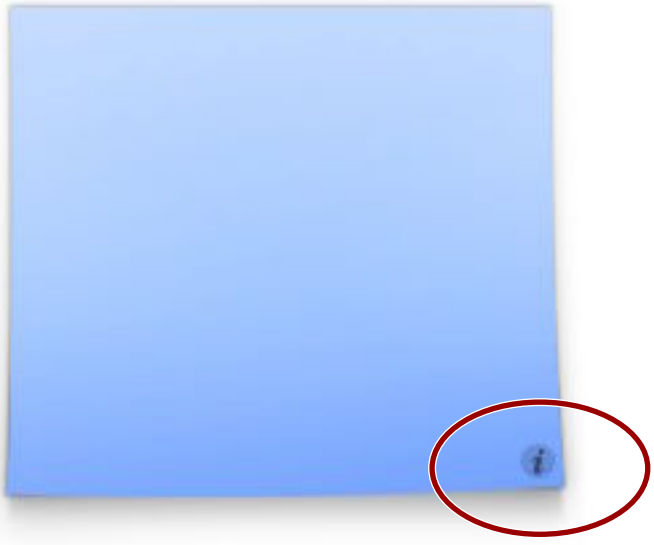


Figure 3-12 The standard info button—users know what this means



- 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 3-13 Proper info button placement



- Use the flip animation only to show your widget's reverse side. The reverse 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 3-14 Aqua controls on a widget's reverse

- 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 reverse of your widget.
- Use a darker or subdued background color for your widget's reverse side. Reusing the background color from your main interface is not advised because it leads to confusion about which side is the main interface.

Figure 3-15 Different backgrounds distinguish between front and reverse

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

Figure 3-16 Branding is appropriate on a widget's reverse side

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

Other Tips

Follow these tips when designing and implementing your widget.

Drop Shadows

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

50% opacity

90 degree angle from horizontal

4 pixel offset (distance from source)

10 pixel size, using Gaussian blur

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 Web Kit 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.

Reverse Sides 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 Preferences" (page 34) discusses how set up your widget for this. "Providing Preferences" (page 33) discusses saving and retrieving preferences.

Note: Most of the code in this chapter can be found in the "GoodbyeWorld" example in `/Developer/Examples/Dashboard/`.

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

The second method 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.

Displaying Preferences

You may find it prudent to provide an interface for setting preferences. Three parts are needed to display preferences: designing your widget with preferences in mind, providing for the transition to the preferences, and returning to the widget's main interface. First, in the 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 also be hidden):

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

    <div id="front" onmousemove='mousemove(event);'
onmouseout='mouseexit(event);'>

        <img span="backgroundImage" src="Images/Default.png">
        <span id="worldText">Hello, World!</span>

        <div class='flip' id='fliprollie'></div>
        <div class='flip' id='flip' onclick='showPrefs(event);'
onmouseover='enterflip(event);' onmouseout='exitflip(event)';></div>

    </div>

    <div id="back">

        <img class="backgroundImage" src="Images/Default_reverse.png">
        <select id='worldPopup' onchange='changeWorld(this);'>
            <option value=1>Hello, World!</option>
            <option value=2>Goodbye, World!</option>
        </select>
        <div class="doneButton" onclick='hidePrefs();'>Done</div>

    </div>

</body>
```

This code sets up the front and back parts of the widget. To hide your back layer, you need to define it as hidden by using a style. Then two layers are defined, using the `<div>` tag. Elements are assigned on each layer to call the show and hide preferences functions. These functions control the display and transition to and from the preferences. The following function switches to the reverse side:

```
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 calls this function, which causes the preferences 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.

Hiding the preferences and returning to your main user interface follows a similar procedure:

```
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.

The Info Button

Widgets designate that they have preferences by displaying an info button in the bottom-right corner of the widget. If your widget has preferences, it should follow this convention.

Note: The info button is available as part of the standard Widget Resources available in `/System/Library/WidgetResources/`. Link directly to the images there instead of making copies.

The info button also fades in and out when the mouse is over the widget. This code does the animation for you:

```
var flipShown = false;

var animation = {duration:0, starttime:0, to:1.0, now:0.0, from:0.0,
firstElement:null, timer:null};

function mousemove (event)
{
    if (!flipShown)
    {
        if (animation.timer != null)
        {
            clearInterval (animation.timer);
            animation.timer  = null;
        }

        var starttime = (new Date).getTime() - 13;

        animation.duration = 500;
        animation.starttime = starttime;
        animation.firstElement = document.getElementById ('flip');
        animation.timer = setInterval ("animate();", 13);
        animation.from = animation.now;
        animation.to = 1.0;
        animate();
        flipShown = true;
    }
}

function mouseexit (event)
{
    if (flipShown)
    {
        // fade in the info button
        if (animation.timer != null)
        {
            clearInterval (animation.timer);
            animation.timer  = null;
        }

        var starttime = (new Date).getTime() - 13;

        animation.duration = 500;
        animation.starttime = starttime;
        animation.firstElement = document.getElementById ('flip');
        animation.timer = setInterval ("animate();", 13);
        animation.from = animation.now;
        animation.to = 0.0;
        animate();
        flipShown = false;
    }
}
```

```
function animate()
{
    var T;
    var ease;
    var time = (new Date).getTime();


    T = limit_3(time-animation.starttime, 0, animation.duration);

    if (T >= animation.duration)
    {
        clearInterval (animation.timer);
        animation.timer = null;
        animation.now = animation.to;
    }
    else
    {
        ease = 0.5 - (0.5 * Math.cos(Math.PI * T / animation.duration));
        animation.now = computeNextFloat (animation.from, animation.to, ease);
    }

    animation.firstElement.style.opacity = animation.now;
}

function limit_3 (a, b, c)
{
    return a < b ? b : (a > c ? c : a);
}

function computeNextFloat (from, to, ease)
{
    return from + (to - from) * ease;
}
```

In your HTML file, you need to specify two of these functions as event handlers, so that these functions can be called when the mouse is over your widget:

```
<div id="front" onmousemove='mousemove(event);' onmouseout='mouseexit(event);'>
```

Note that these handlers exist only on the front of the widget. To exit your preferences, use a Done button. This affirms to the user that the preferences set have been accepted.

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:

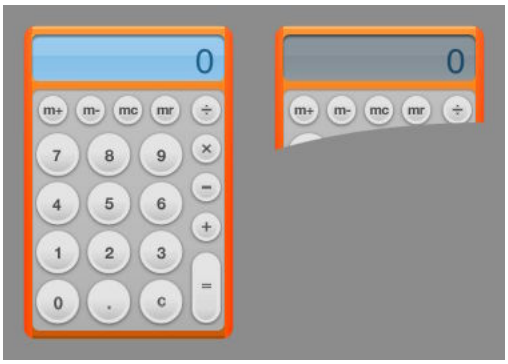
```
function onhide () {
    if (timer != null) {
        clearInterval(timer);
        timer = null;
    }
}
```

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 5-1 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.ondragstop`

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.ondragstop = widgetDragStopHandler;
```

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;
```

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 (Figure 3-1) provides an example of how to create control circles and rectangles. The highlighted regions are the areas from which dragging is not allowed.

Figure 6-1 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 6-1 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 6-2 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:

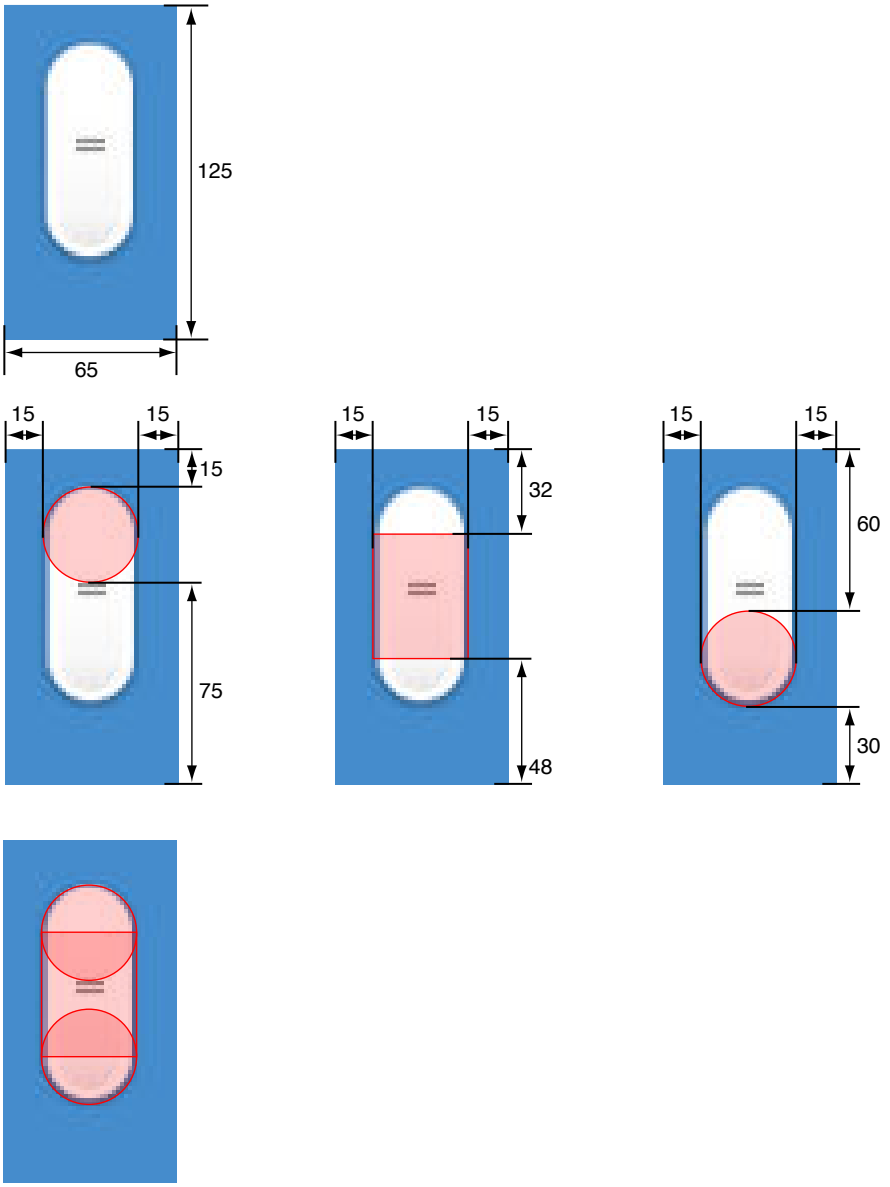
```
.control-circle-example {
    ...
    -apple-dashboard-region: dashboard-region(control circle 5px 5px 5px 5px);
    ...
}
```

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 32x 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 6-2 (page 45).

Figure 6-2 Control region example



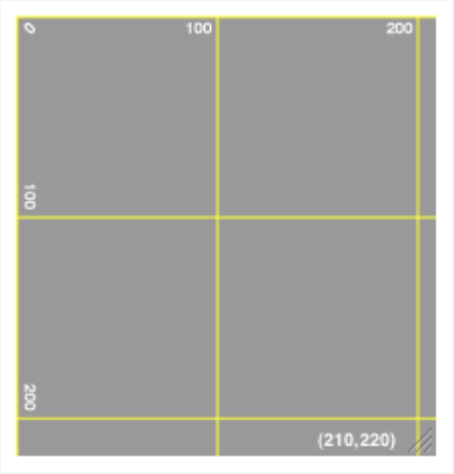
Note that the circle regions are centered within the given bounds.

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

Resizing

Widgets 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: Most of the code in this chapter can be found in the “Grid” example in `/Developer/Examples/Dashboard/`.



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;
    right: 2px;
    bottom: 2px;
    -apple-dashboard-region: dashboard-region(control rectangle);
}
```

In your JavaScript file, include this code:

```
var lastPos;

function mouseDown(event)
{
    var x = event.x + window.screenX;
    var y = event.y + window.screenY;

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

    lastPos = {x:x, y:y};

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

function mouseMove(event)
{
    var screenX = event.x + window.screenX;
    var screenY = event.y + window.screenY;

    window.resizeBy(screenX - lastPos.x, screenY - lastPos.y);
    lastPos = {x:screenX, y:screenY};

    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 larger 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 `setCloseBoxInset` 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.

Localization

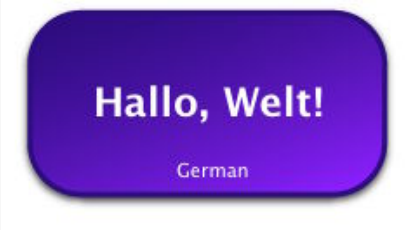
Localizing your widget provides a more comfortable and pleasant experience for foreign language speakers. If your widget is used in areas where languages other than English is spoken, you should localize it.

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 "HelloWelt" example in `/Developer/Examples/Dashboard/` provides sample code on localizing 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 "Widget Bundle Structure" (page 15).

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 8-1 (page 52) lists of common languages and their corresponding language project directory names:

Table 8-1 Common languages and corresponding language project names.

Language	Language project directory name
Chinese (Simplified)	`zh_Hans.lproj`

Language	Language project directory name
Chinese (Traditional)	zh_Hant.lproj
Danish	da.lproj
Dutch	nl.lproj
English	en.lproj
Finnish	fi.lproj
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*.

Security

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

Widget Security Model

Using certain resources within your widget may pose a security risk for users. In these circumstances, the widget security model provides a method for Dashboard to be aware that your widget may perform insecure tasks. If your widget is working with resources that pose a security threat to the user, the user must approve before access is granted.

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

- Access files outside of your widget bundle
- Use a Web Kit or standard browser plug-in
- Access network resources
- Run a Java applet
- Run a command-line utility
- Using 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 9-1 (page 57):

Table 9-1 Info.plist Keys for the Widget security model

Key	Type	Definition	Example
`AllowFileAccessOutsideOfWidget`	Boolean	Access to files across the file system; limited by the user's permissions.	`<img src="~/Sites/images/macosxlogo.gif">`

Key	Type	Definition	Example
`AllowFullAccess`	Boolean	Access to the file system, Web Kit and standard browser plug-ins, Java applets, network resources, and command-line utilities.	N/A
`AllowInternetPlugins`	Boolean	Access to Web Kit 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.createScript ("/usr/bin/foo", null, null);`
`Plugin`	String	Specifies a widget plug-in.	`foo.widgetplugin`

If any of these keys are present in your information property list file and it's located outside of `/Library/Widgets/`, a dialog is presented to users upon your widget's first load. The dialog asks them whether or not they want to use your widget. If the request is approved, your widget is loaded and granted access to the resources that it requested. The request is not repeated on subsequent loads if approved. If the request is denied, your widget is not allowed to load. If your widget is loaded again, the request is made to the user again.

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

External Access

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 "Security" (page 57) to learn more about the widget security model.

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:

```
<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>
```

Here, an arbitrary function is called when a user clicks within some text. A portion of a URL is passed to the 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>
```

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 "Command-Line Access" (page 61), with the `open` command-line utility
- Implement a widget plug-in, as discussed in "Widget Plug-in" (page 67)

Command-Line Access

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 "Security" (page 57) to learn more about the widget security model.

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 11-1 `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.

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.

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 it's 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 11-2 `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;`

Sample Code

The widget sample code in `/Developer/Examples/Dashboard/` includes the Which widget. It's a simple example the wraps a widget front end around the `which` command line utility.

The widget's action button calls a method, which in turn uses `widget.system()` to call which on a provided string and then place it's output string back in to the widget:

```
document.getElementById("outputText").value = widget.system("/usr/bin/which "+
 document.getElementById("inputText").value, null).outputString;
```

Note that, as previously mentioned, that execution of code in the widget waits for the result of `widget.system()`. For simple commands such as `which`, this is fine. Other commands may take longer to execute or rely on input within their execution. For these, use the asynchronous operation mode of `widget.system()`.

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.

Because the command is running asynchronously, it's 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 11-3 `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).

Option	Purpose	Description
`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 outHandler(currentStringOnStdout){    // Code that does something with
 the command's current output like...
    document.getElementById("element").innerText = currentStdout;}
```

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 code in `/Developer/Examples/Dashboard/` 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
                    );
    }
}
```

Widget Plug-in

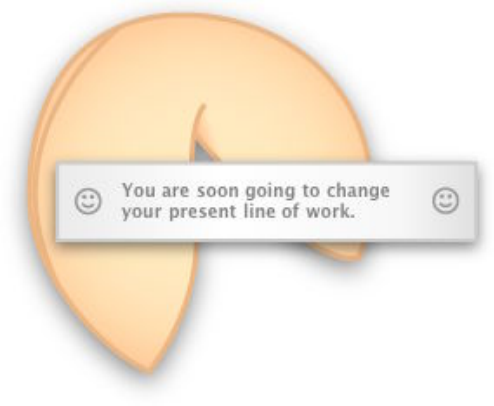
Widgets alone cannot access applications directly, receive distributed notifications, or read files from disk. To enable these interactions, you need to provide a plug-in. You are required to implement an interface for your plug-in that makes itself available to the widget. This interface communicates 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.

Note: Before reading this chapter, read "Security" (page 57) to learn more about the Widget security model.

Note: The "Fortune" example in `/Developer/Examples/Dashboard/` provides sample code on creating a widget plug-in.



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. 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 `NSPrincipleClass` 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. Having this value present also activates the Dashboard Security Model, as detailed in "Security" (page 57).

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.

Widget Delivery

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 installed. When downloaded using other web browsers users need to install the widget manually.

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, show Dashboard, click the Plus sign to display the Widget Bar and click the widget's icon in the Widget Bar to open it. If you're using a browser other than Safari, click the download link.

When the widget download is complete, unarchive it and place it in /Library/Widgets/ in your home folder. show Dashboard, click the Plus sign to display the Widget Bar and click the widget's icon in the Widget Bar to open it.

Document Revision History

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

Date	Notes
Tiger	Includes information about distributing widgets and more widget design guidelines.
	Updated for public release of Mac OS X v10.4. First public version.
	Reorganized the document into task-based chapters. Includes new chapters on the widget security model, command-line access, design guidelines, and overview. Also added sections on drag events and the widget close box. Added more sample code.
2004-11-02	Revised Widget operations and widget plug-in. Relocated Canvas, Pasteboard, and Drag and Drop chapters to Safari JavaScript Programming Topics and included links.
2004-10-04	Added Localization and Application Activation topics. Revised Canvas, Control Regions, Widget Resizing, Preferences, and more topics.
2004-08-31	Fixed code and formatting issues.
2004-06-28	New document that provides an overview of the Dashboard environment and the widgets that exist in it.

Document Revision History