

# *From Desktop to Consumer Devices*

## *The Applet Writer's Style Guide* *Draft—Version 0.8*

Please send comments on this draft version to  
[personaljava-comments@java.sun.com](mailto:personaljava-comments@java.sun.com)



*Sun*

microsystems

JavaSoft  
2550 Garcia Avenue  
Mountain View, CA 94043 U.S.A.  
408-343-1400

December 1997

## Copyright Information

© 1997, Sun Microsystems, Inc. All rights reserved.  
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

This document is protected by copyright. No part of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

The information described in this document may be protected by one or more U.S. patents, foreign patents, or pending applications.

## TRADEMARKS

Sun, Sun Microsystems, Sun Microelectronics, the Sun Logo, SunXTL, JavaSoft, JavaOS, the JavaSoft Logo, Java, HotJava Views, HotJavaChips, picoJava, microJava, UltraJava, JDBC, the Java Cup and Steam Logo, "Write Once, Run Anywhere" and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

UNIX<sup>®</sup> is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

Adobe<sup>®</sup> is a registered trademark of Adobe Systems, Inc.

Netscape Navigator<sup>™</sup> is a trademark of Netscape Communications Corporation.

All other product names mentioned herein are the trademarks of their respective owners.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE DOCUMENT. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.



Please  
Recycle

# *Table of Contents*

---

<b>Preface</b> .....	<b>5</b>
Who should read this .....	5
Changes since previous version .....	5
<b>Overview</b> .....	<b>5</b>
<b>Human Interface Design</b> .....	<b>6</b>
The Consumer Audience .....	6
Keep It Simple .....	7
Immediate and Continuous Feedback .....	7
Capturing the User's Attention .....	8
Interface Elements to Avoid .....	8
Unsupported Interface Elements .....	12
<b>Input/Output Considerations</b> .....	<b>12</b>
Input Devices .....	13
Graphics Considerations .....	18
General Graphics Considerations .....	18
Television-Specific Display Considerations .....	22
Typeface Considerations .....	23
<b>Resource Limitations</b> .....	<b>24</b>





## *From Desktop to Consumer Devices* *The Applet Writer's Style Guide*

### *Preface*

#### *Who should read this*

This document is for people used to writing applets or applications for desktop computers or workstations who want to take advantage of the large audience emerging in consumer electronic devices by using PersonalJava. It focuses on the interaction with the end user by an applet running on a device with limited resources and assumes the end user has no technical experience.

#### *Changes since previous version*

As compared to version 0.7, discussion of `sunw` (`sun.com`) was removed. A paragraph describing testing for the presence of `isdoublebuffered` by reflection was added as was an example of using `Timer`.

### *Overview*

PersonalJava is a new Java platform adapted for consumer electronics, which is integrated with the Java Developer's Kit (JDK). It contains new, device-specific features such as the Personal AWT (Abstract Windows Toolkit), a modification of the Java AWT tuned for consumer product look and feel and for resource constraints.

The market for small consumer electronics devices, which includes devices such as screenphones and set-top boxes, is far larger than the market for personal computers. PersonalJava enables you to reach this large audience with your software products.

Writing software for consumer electronics differs significantly from writing for workstations and desktop computers. If you are accustomed to writing for desktop machines, this style guide introduces you to some of the special considerations necessary for consumer electronics devices. Among the most important of these considerations are the limitations of the hardware and the experience and expectations of the user:

- *The hardware* — Consumer electronics devices have less memory and different input and output devices than typical personal computers. You need to know how to deal with the limited resources and unfamiliar I/O devices of the platforms on which your software runs.
- *The users* — Many users of consumer electronics devices are unfamiliar with desktop computers. They may not be acquainted with concepts such as popup menus, scroll bars, or the computer desktop. They may feel uncomfortable dealing with any device they consider to be too “high-tech” and tend to be unwilling to learn complex models. On the other hand, they are familiar with devices such as TV remote controls, push-button phones, and the control panels on microwave ovens. You need to make your software interfaces as non-intimidating, familiar, and easy-to-learn as possible.

The suggestions that follow will help you to construct your applets to meet the special requirements of consumer devices. They include guidelines for designing an effective interface, helpful hints for dealing with resource limitations, warnings about potential pitfalls, and example code.

## *Human Interface Design*

### *The Consumer Audience*

Electronic devices such as televisions, VCRs, telephones, and microwave ovens are common today. However, not all such products gain wide acceptance. Eight out of ten consumer products fail in the marketplace, often because consumers find them too difficult to use. Modern consumers have little patience for learning how to operate new products. Most consumers simply will not buy a device if they believe it might not be easy to use. When consumers see a product they can't easily figure out, they're likely to say, “It should just work!”

## *Keep It Simple*

The most important guideline for consumer interfaces is “Keep It Simple.” This rule applies to everything the user sees, hears, or touches on a product. It should guide the design of everything from a product’s appearance to the way users navigate among a product’s functions. Simple interfaces are a common feature of most successful consumer devices. Consider, for example, the push-button interface that programs a microwave oven.

Simple interfaces are especially important for consumer electronics devices for two reasons:

- Consumers are not motivated to learn complicated methods for performing operations.
- The input devices on many consumer electronics devices can’t support complex operations.

Here are some suggestions for keeping your interface simple:

### ***Use a single-click interaction model.***

A single click or similar action should be the basic activator. To the extent possible, the user should be able to access everything in your interface through simple actions such as a single click of a button or tap of a finger; see “Input Devices” on page 13.

### ***Minimize the number of interface elements.***

An interface element is any feature of your product that interacts with the user. The complexity of an interface increases with the number of elements involved, especially those that require the user to fill in information or make choices. Keep the number of your product’s interface elements to a minimum. In addition to minimizing the number of interface elements, you should try to avoid some types of interface elements altogether.

## *Immediate and Continuous Feedback*

Users of consumer devices expect immediate response to their input. When response is not immediate, consumers often become annoyed with the product — or worse, assume the device is broken.

The interface of your applet should respond continuously. This must be true even in cases where completion of an operation is delayed, as for example,

when it is aware that information is being downloaded. Displaying a static status message is not adequate. Your applet should provide dynamic, live feedback to the user during any operational delays.

## *Capturing the User's Attention*

When designing applets for use by consumers, make important information stand out. For every display, decide which piece of information is most important and make that the focus of the user's attention.

### ***Minimize the choices users must execute to navigate.***

Where possible, guide the user through your applet with clearly labeled buttons and graphics. Assume your applets are running on a small screen.

Minimize a user's need to scroll your pages. Assume only vertical scrolling is available, which the user commonly controls with plastic scroll buttons.

## *Interface Elements to Avoid*

The following interface elements characteristically cause problems for consumer users.

### ***Avoid multiple overlapping windows***

Multiple overlapping windows (see Figure 1) are very confusing to consumer users and require a much steeper learning curve than is appropriate for most consumer electronics devices. Moreover, they are unavailable in many environments; see "Measure AWT components." on page 19. The same concerns apply to drag-and-drop (Figure 2).

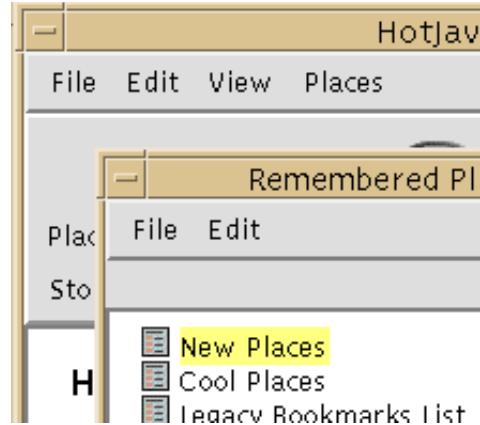


Figure 1 Don't create a confusing visual field with overlapping windows.

### ***Avoid drag and drop.***

The mouse press and release or similar actions involved in drag and drop is confusing and hard to learn for consumer users and most consumer devices do not support it.

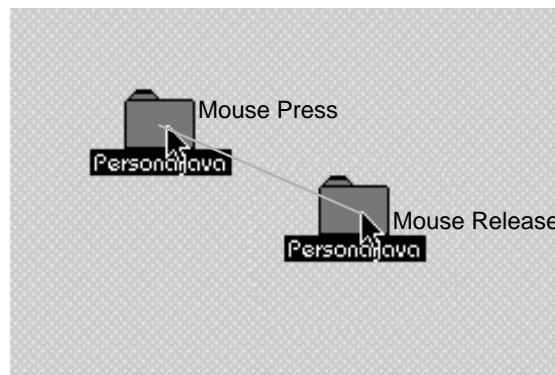


Figure 2 Don't confuse users with drag and drop.

### ***Avoid hierarchical menus.***

Hierarchical menus extend a sub-menu from an item on a higher-level menu. (Figure 3). They require additional learning and can be confusing to consumer users. Moreover, hierarchical menus are sometimes implemented by holding

down a button and moving the cursor, a procedure most consumer electronics devices do not support.

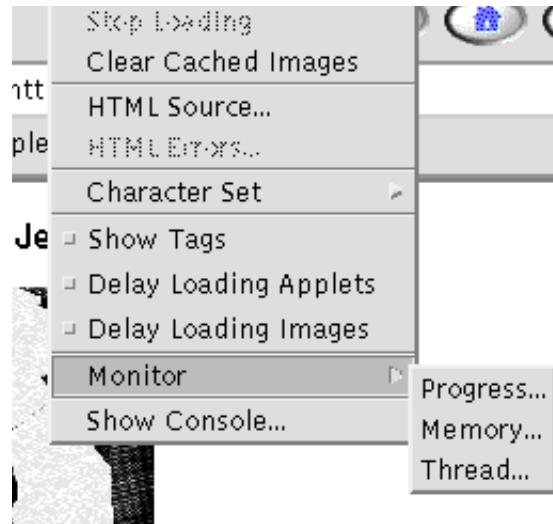


Figure 3 Don't confuse users with hierarchical menus.

### ***Avoid double clicking.***

Never use double clicking as a base method for an operation. Most consumer products do not have a mouse or any device that provides for double clicking; see "Pay particular attention to simulated input devices." on page 17. Even when consumer products do have mice, the pointing device may inadvertently move between clicks causing the system to interpret the double-click as two single clicks. Moreover, users with common medical conditions like arthritis or Repetitive Stress Syndrome (RSI) may be even less tolerant of double clicking in a consumer electronics environment than in a desktop environment.

Double tapping on a touch screen presents further problems because people don't consistently hit the same spot on the screen twice in rapid succession. What the user meant as a double tap may get parsed by the input handler as two discrete single taps, and then the user sees the software do the wrong thing twice instead of the right thing once.

### ***Use popup menus with caution.***

If you must use a popup menu, invoke it from a specific popup graphic on the screen. You should not use hidden popup menus that are activated by clicking

anywhere on the display. Popup menus should be clearly labeled and should always show their current state. The implementation should support the ability to display the menu with a single click or the equivalent and allow the user to choose a menu item with a single subsequent click.

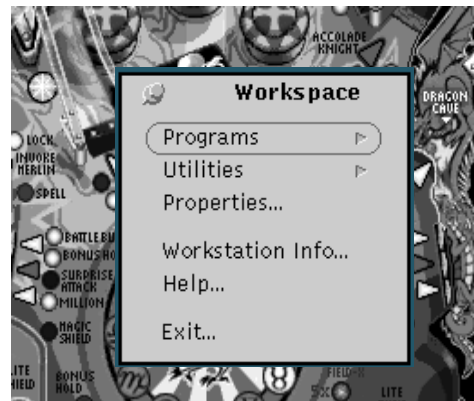


Figure 4 Don't obscure an interface with a popup menu.

### ***Use modal dialogs with caution.***

Dialogs are windows that depend on other windows, and modal dialogs are such secondary windows that block interaction with the system until the user dismisses them (Figure 5). PersonalJava supports the Java Dialog API in modified form, but be cautious with dialogs. Users become confused if they click (or the equivalent) in accustomed places and nothing happens. If you

must use a modal dialog, there should be only one modal dialog on the screen at a time. If you use a modal dialog to alert the user to an error condition, you should take care to present the condition in a way that does not panic the user into thinking the device is broken.

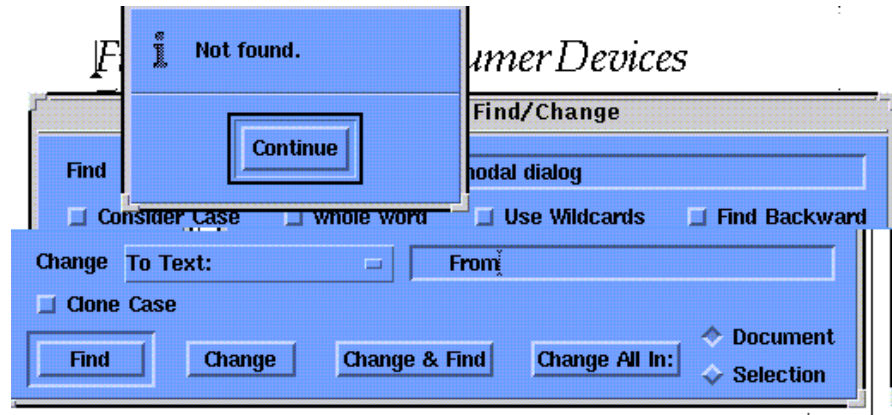


Figure 5 Don't block interaction with the system by requiring a user to dismiss a modal dialog.

## *Unsupported Interface Elements*

The classes that support some interface elements in the Java API are absent from the PersonalAWT. These interfaces include file selection boxes and hierarchical menus of all varieties. Overlapping windows are supported only on some platforms.

You might recreate the unsupported functions by programming them from scratch, but that is not advisable because typical consumer devices don't support them and because they can be confusing to consumers.

## *Input/Output Considerations*

This section discusses how input and output capabilities of various platforms affect applet design.

## *Input Devices*

Input devices vary widely among different platforms. The consumer may interact with your applet by using a remote control with specialized buttons, a specialized keyboard, a joystick, a touch screen, or some other collection of buttons. Most of these input devices are familiar and do not require discussion here. The exception is the remote control, which may use a combination of *selection boxes*, *selection cursors*, *Go operations*, and *virtual keyboards* to select areas of the applet:

- A **selection box** moves around on the screen like keyboard-based focus traversal on a desktop system. The selection box selects an entire component such as a text field or a button.
- A **selection cursor** moves within a specific component. The selection cursor specifies a single *x,y* coordinate within the component. The selection box and selection cursor are never used at the same time.
- A **Go operation**, initiated via a button on the remote control, performs a context-sensitive operation such as selecting an HTML link, activating a button, or displaying a virtual keyboard.
- A **virtual keyboard** is an on-screen representation of a keyboard. The user maneuvers a cursor or similar element to a key and then clicks to enter the symbol.

Although different platforms employ different strategies for accommodating their input devices, the following suggestions generally will improve the usability of your applet.

### ***Control User Access.***

Some systems allow only one type of interaction with a component. For example, viewers may not be able to use a virtual keyboard if they are also selecting *x,y* coordinates within a component. You control access with the input preference APIs: `NoInputPreferred`, `ActionInputPreferred`, `KeyboardInputPreferred`, and `PositionalInputPreferred`, which are part of `com.sun.awt`.

In mouseless environments, users typically can navigate from one on-screen component to another by using keys or buttons on the system's input device. When the user navigates to a component, the visual representation of that component is modified in some way to indicate that it is the current component.

The input device typically provides a way for the user to select the current component. For example, after navigating to an on-screen button component, the user might press the Go key on a remote control to “press” the on-screen button.

PersonalJava provides input preference interfaces that let you specify how users navigate among components and how they interact with them, which are called the input preference API's. These interfaces are:

- `NoInputPreferred`. The user may not navigate to the component. This API might be appropriate for components such as labels.
- `KeyboardInputPreferred`. The user can make keyboard input. Some platforms respond by popping up an on-screen keyboard when the user selects this component.
- `ActionInputPreferred`. The user can click on or otherwise activate the component by using the input device.
- `PositionalInputPreferred`. The user can select x,y coordinates within the component.

For example, class `MyLabel` represents an area that simply displays a text string. It isn't concerned with any sort of input at all, so it implements `NoInputPreferred`.

```
public class MyLabel extends Canvas implements
NoInputPreferred {

    public MyLabel (String text) {
        ...
    }

    public void paint(Graphics g) {
        g.drawString(text, x, y);
    }
}
```

Class `MyNumberField` represents an area where the user can type a number. It concerns itself only with character input, not with positional or other sorts of input and employs `KeyboardInputPreferred`.

```
public class MyNumberField extends Canvas implements
KeyboardInputPreferred, KeyListener {
```

```
public MyNumberField() {
    ...
}

/* Insert the key in the field */
public void keyTyped(KeyEvent e) {
    ...
}

/* Ignore it - input handled by keyTyped */
public void keyPressed(KeyEvent e) {
}

/* Ignore it - input handled by keyTyped */
public void keyReleased(KeyEvent e) {
}

/* We want the user to be able to TAB to this
   component. */
public boolean isFocusTraversable() {
    return true;
}

...
}
```

Class `MyButton`, below, represents a button on the screen. It only concerns itself with actions (up and down clicks) and whether it's selected, so it implements `ActionInputPreferred`:

```
public class MyButton extends Canvas implements
ActionInputPreferred {

    public MyButton(Image down, Image up, Image disabled) {

        MouseListener mouseListener = new MouseAdapter() {

            public void mousePressed(MouseEvent e) {
                ... // User clicked; do something
            }

            public void mouseReleased(MouseEvent e) {
                ... // User released
            }

            public void mouseEntered(MouseEvent e) {
                ... // User has navigated here
            }
        }
    }
}
```

```
        public void mouseExited(MouseEvent e) {
            ... // User has navigated away
        }
    };

    addMouseListener(mouseListener);
}

...
}
```

Class `MyScribbler` creates an area where the user can draw. It concerns itself with mouse up and down events, keyboard input, and the position of the cursor. Note that because this component has to act more like a drawing pad than a type-in field, the code uses `PositionalInputPreferred` instead of `KeyboardInputPreferred`.

```
public class MyScribbler extends Canvas implements
    PositionalInputPreferred,
    MouseMotionListener, KeyListener
{
    private final static char controlP = 0x10;

    Color currentColor = Color.black;
    boolean painting = true;

    public MyScribbler() {
        ...
    }

    /* User is holding the button while moving. Draw
       a line.
       Since not all input devices allow dragging,
       we will provide other means for doing this in the
       mouseMoved method. */
    public void mouseDragged(MouseEvent e) {
        addLine(e.x, e.y, currentColor);
        repaint();
    }

    /* User has moved the pointing device to a new
       position.
       If the pointing mode, activated by CTRL-P, is on,
       paint the line. */
    public void mouseMoved(MouseEvent e) {
        if (painting) {
```

```
        addLine(e.x, e.y, currentColor);
        repaint();
    }
}

/* Show this character at the current position */
public void keyTyped(KeyEvent e) {
    char ch = e.getKeyChar();
    if (ch == controlP) {
        painting = !painting;
    } else {
        addChar(e.x, e.y, e.getKeyChar());
        repaint();
    }
}

/* Ignore key press; it's handled by keyTyped */
public void keyPressed(KeyEvent e) {
}

/* Ignore key release - it's handled by keyTyped */
public void keyReleased(KeyEvent e) {
}

/* We don't want the user to be able to TAB to this
component. */
public boolean isFocusTraversable() {
    return false;
}
}
}
```

### ***Pay particular attention to simulated input devices.***

Some platforms simulate mouse events. On such devices no way may exist for the user to double click, to specify a mouse button other than the first one, or to press and drag. Do not assume support for:

- `Event.clickCount` being larger than one.
- `Event.modifiers` being non-zero.
- Receiving a `MOUSE_UP` event at coordinates different from, or at a time different than, the preceding `MOUSE_DOWN` event.

Likewise, some systems simulate a keyboard, but, even when they do, some modifier keys may be absent, and no way may exist for the user to hold down a key. Do not depend on any of the following for core functionality:

- `Event.modifiers` being non-zero.
- The interval between a `KEY_RELEASE` or `KEY_ACTION_RELEASE` and a previous `KEY_PRESS` or `KEY_ACTION` event being meaningful.

## *Graphics Considerations*

This section contains guidelines for your applet's graphics.

### *General Graphics Considerations*

#### ***Keep your applet size flexible.***

Because of the variation in usable display area, there is no guarantee that a browser will be able to honor the `WIDTH` and `HEIGHT` attributes on the `APPLET` tag. The pixel dimensions of screens on telephones can vary from 320 x 240 to 160 x 480, for example. On some platforms, increasing the size of the applet window may reduce the amount of memory available to it; do not make your applet larger than necessary. You are strongly encouraged to use percentage values for your `WIDTH` and `HEIGHT` attributes (`WIDTH` is defined as a percentage of the browser's display width and `HEIGHT` is defined as a percentage of the browser's display height). For instance, use:

```
<applet . . . WIDTH="50%" HEIGHT = "25%" . . .  
</applet >
```

instead of:

```
<applet . . . WIDTH="160" HEIGHT = "120" . . .  
</applet >
```

#### ***Avoid small typeface sizes in graphics.***

Some applets use images (such as GIF or JPEG images) that already have text drawn inside them. Small text inside an image is often unreadable on low-resolution devices. Bear in mind that television viewers often sit fifteen feet from their screen. Avoid such images whenever possible. A better solution is to render the text with `drawText`.

## ***Be aware of font metrics.***

You can't make any assumptions about the size of a font's characters based on the design size (the point size, as specified in `java.awt.Font`). Always measure text using the `java.awt.FontMetrics` class, which returns the size in pixels. Be aware that the ascent of some characters may be larger than the number returned by `getAscent`. You can use `getMaxAscent` to get the maximum ascent of the font.

## ***Do not depend on subtle color differences.***

Some devices use as few as 4 or 16 colors (or shades of gray). Similar colors may blend into one in your graphics or components.

## ***Do not use XOR-mode drawing.***

There is no reliable way of correctly implementing XOR-mode drawing on an anti-aliased display system. The resulting drawings may be invisible to the user.

## ***Measure AWT components.***

AWT components may have radically different sizes on different platforms. For example, platforms with low-quality output devices (such as televisions) often magnify text. This can cause components that contain text to be larger than non-text components. When possible, use layout managers to ensure that the size of the layout varies with the platform. If you are using absolute layout, always use calls such as `getSize` and `getMinimumSize` to determine the size of the components that you are positioning.

On some platforms, the window system may refuse to allow the user to interact with any component that is obscured in any fashion (for example, components that are clipped by a parent, a sibling, or a child). If you fail to measure components properly and then lay them out incorrectly, users may be unable to interact with your applet.

## ***Allow for size to change when text changes.***

When you change the text in a component, as with `Button.setText` for example, the component may need to be resized. You should call `invalidate` on the component and `validate` on the component's container after changing the text.

The following Paint example applet creates a label and a button. Each time you press the button, it adds an extra 'X' onto the end of the label. Without the calls to `invalidate` and `validate`, the extra 'X's would quickly run off the end of the label.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;

public class TextResize extends Applet implements
ActionListener{

    Label label;
    Button button;
    public void init() {
        label = new Label("A label");
        button = new Button("A button");
        button.addActionListener(this);
        add(label);
        add(button);
    }

    public void actionPerformed(ActionEvent ex) {
        label.setText(label.getText() + "X");
        invalidate();
        validate();
    }
}
```

### ***Implement the paint and update methods carefully.***

Many applets omit updating their components correctly when `paint` or `update` is called. This serious failure leads to user-visible problems on some platforms that may make your applet unusable.

Do not assume that calls to `paint` are infrequent. Applets that do not repaint themselves properly run the risk of acquiring visual defects, even on traditional desktop platforms. In addition, some platforms do not store the contents of a component's window in a reusable form, and therefore may need to repaint the entire window when any change is made to it (including changes made by the applet itself).

Draw on-screen components only within calls to `paint` or `update`. Though it may be tempting to call `getGraphics` on a component once, save the result, and draw it later, this technique is not reliable or safe. Instead, call `repaint`

to schedule a call to `update`. Note that a call to `getGraphics` on an off-screen image created by `createImage(int, int)` is safe. Only a call to `getGraphics` on a component is risky.

As examples, here are two scenarios involving `paint` and `update`, with recommendations for avoiding problems.

- An author writing an animation applet might think it is OK to use a timer thread to update itself several times a second. The author incorrectly believes that because the timer thread does redrawing so frequently, proper implementation of `paint` is unnecessary. However, such an applet may fail on some platforms.

Instead, the timer thread should call `repaint` to request an immediate call to `update`. The following code illustrates the proper method:

```
import java.awt.*;
import java.applet.*;

public class Anim extends Applet implements Runnable {
    Image[] images;
    int curFrame = 0;
    Thread timerThread = null;
    public void init() {
        images = new Image[2];
        images[0] = getImage(getCodeBase(), "image1.gif");
        images[1] = getImage(getCodeBase(), "image2.gif");
    }
    public void paint(Graphics g) {
        update(g);
    }
    public void update(Graphics g) {
        g.drawImage(images[curFrame], 0, 0, this);
    }
    public void start() {
        timerThread = new Thread(this);
        timerThread.start();
    }
    public void stop() {
        timerThread = null;
    }
    public void run() {
        while (timerThread == Thread.currentThread()) {
            try {
                Thread.sleep(300);
            } catch (InterruptedException ex) {
                return;
            }
            int nextFrame = curFrame + 1;
```

```
        if (nextFrame == images.length) {
            nextFrame = 0;
        }
        curFrame = nextFrame;
        repaint();
    }
}
public void destroy() {
    for (int i = 0; i < images.length; i++) {
        images[i].flush();
        images[i] = null;
    }
    images = null;
}
}
```

- An author wants an applet to scroll the contents of a component. Inside the `paint` method, the author draws the contents, calls `Thread.sleep`, and then redraws the contents at a different location. This is a bad practice for at least two reasons. First, any operation that may take a while (such as `Thread.sleep`, `wait`, or network operations) should not be performed in `paint` or `update`, because doing so may block other components from being updated, and, in some implementations, may prevent other applet-related events from happening. Second, you can't assume that any intermediate stages in `paint` or `update` are visible to the user because some platforms batch-up drawing requests and may not perform them until the `paint` or `update` method exits. Other platforms may perform implicit double buffering, copying the off-screen buffer into a user-visible location only after `paint` or `update` returns.

### ***When you call `repaint`, specify the exact coordinates of the affected area.***

When you call `repaint` to request an update to your component and you know that only a small section has changed, specify the exact coordinates of that section. This precaution can radically improve the refresh performance.

### ***Television-Specific Display Considerations***

The television environment has some unique requirements. Here are some recommendations for improving the quality of applets when they appear on a TV screen.

## ***Limit yourself to the usable area of the screen***

The usable area of a television screen in terms of pixels, what is called the safe title zone, is smaller than the physical area and depends on the device involved. Some developers use 544 by 378. You should try to inform yourself of the safe title area in the environment you are working in. You can use `Toolkit.getDefaultToolkit().getScreenSize` to determine the physical display area available.

## ***Use light-colored text against dark backgrounds.***

Viewers find it easier to read light-colored text against a dark-colored background. `Color.darkGray` (`Color(64, 64, 64)`) is a good background.

## ***Don't use full white, yellow, or red backgrounds.***

Although many applets use full white backgrounds, this is not the best choice if you want your applet to display well on a television. Full white, yellow, or red backgrounds cause screen distortion in which the edges of images appear to bow. If you want a white background, try using 90% white (`Color(230, 230, 230)`).

## ***Avoid single-pixel horizontal lines.***

Televisions have problems displaying single-pixel horizontal lines. Instead, use lines that are at least two pixels wide.

## ***Typeface Considerations***

### ***Do not assume specific type faces are available.***

Many logical typeface names may map to the same typeface. The most you can assume is that `SansSerif` (formerly “Helvetica”) and `Serif` (formerly “TimesRoman”) map to a proportional-width typeface and that `Monospaced` (formerly “Courier”) maps to a fixed-width typeface.

### ***Use only ISO Latin 1 characters.***

PersonalJava implementations typically render fewer glyphs than do desktop platforms. You should limit your selection to ISO Latin 1 characters, which are likely to be present on all platforms.

Use of non-Latin 1 characters may create problems on some platforms. For instance, this code fragment may not display anything, even though it uses a plain, sans serif font:

```
// Output some Chinese ideograms in sans serif
g.setFont(new Font("SansSerif", Font.PLAIN, 20));
g.drawString("\u4E00\u4E01\u4E02", 10, 10);
```

## Resource Limitations

Small consumer electronics devices are resource-limited. PersonalJava applets must run in environments with relatively little memory. The number of threads per applet may be limited, heap space may be limited, and other components that require large resources may not work. This section offers some guidelines to help you make the best use of the resources available.

### **Use URL-based images instead of off-screen images.**

When possible, use URL-based images (via `Applet.getImage` or `Toolkit.getImage`) instead of either off-screen images (via `Component.createImage(int, int)`), or `ImageProducer`-based images (via `Component.createImage(ImageProducer)`).

### **Use Automated Double Buffering for Animation.**

If you are using animation or need smooth (non-flickering) screen updates for any reason, use the double-buffering API provided by PersonalJava, (`java.awt.Component.isDoubleBuffered`), which returns a boolean stating whether or not the platform performs double-buffered update.

If `isDoubleBuffered` returns true, then all drawing done inside paint and update methods is double buffered automatically.

The default value for the double buffering setting is platform-specific.

If `isDoubleBuffered` returns false and you need non-flicker updates, you must explicitly create an off-screen image, draw into it, and use `drawImage` to display it. However, doing so may take large amounts of memory and could therefore fail.

Here is an example of using the double buffering interface. In this example, an applet class `MyAnimator` checks at initialization to see if the hardware

supports automatic double buffering of its graphic output. If not, it allocates its own off-screen image buffer.

At the point marked by a marginal note, this code uses reflection to test whether the `Component` class has the `isDoubleBuffered` method. This method exists only in PersonalJava and in versions of the JDK numbered 1.2 and above, so, if you are developing code with the JDK 1.1 or expect it to run in JDK 1.1 environments, you need to test for the presence of this method before trying to invoke it. Moreover, in order for your code to compile on JDK 1.1 platforms where the method doesn't exist, you also have to use reflection to invoke it, as shown here.

```
public class MyAnimator extends Applet implements
Runnable {
    ...
    Image offScreenBuffer = null;
    Graphics offScreenGraphics = null;
    void allocateOffScreen() {
        // Create an off-screen Image buffer and a
        // Graphics context
        offScreenBuffer = createImage(getSize().width,
        getSize().height);
        offScreenGraphics = offScreenBuffer.getGraphics();
    }
    public void init() {
        // If the hardware doesn't handle double-
        // buffering...
        try {
            java.lang.reflect.Method m =
                myComponent.getClass().getMethod
                ("isDoubleBuffered", null);

            if (((Boolean)m.invoke(myComponent,
            null)).booleanValue()) {
                allocateOffScreen();
            } catch (Throwable oops) {

            }

            ...
        }
        public void paint(Graphics g) {
            if (offScreenGraphics == null) {
                // Hardware handles double-buffering, so just
                // draw on the screen
                drawFrame(g);
            } else {
                // We have to do double-buffering ourselves.
            }
        }
    }
}
```

Reflection test for double  
buffering



```
        // First draw off screen into allocated image
        // buffer.
        Dimensions d = getSize();
        if (sizeDiffers(this, offScreenBuffer)) {
            allocateOffScreen();
        }
        drawFrame(offScreenGraphics);
        // Then copy the buffer onto the screen.
        g.drawImage(offScreenBuffer, 0, 0, this);
    }
    ...
}
```

## **Use the PersonalJava Timer API for Animation.**

PersonalJava supplies a timer API. The following sample code shows the use of this API to time some event every 200 milliseconds, for example to time an animation.

Unless it has unusual timing requirements, an applet should use the timer provided by `getTimer` rather than creating one of its own. By doing so, an implementation may be able to conserve resources by providing timing services via an existing thread rather than creating a new one.

```
import java.applet.Applet;
import com.sun.util.*; // This is where the timer code is
// located.

public class TimerApplet extends Applet implements
PTimerWentOffListener {
    PTimerSpec myTimerSpec = new PTimerSpec();
    PTimer myTimer = PTimer.getTimer();

    public void init() {
        // Fill in the specification for the timer.
        myTimerSpec.setRepeat(true);
        myTimerSpec.setTime(200);
        myTimerSpec.addTimerWentOffListener(this);
    }

    public void start() {
        ...
        myTimer.schedule(myTimerSpec);
        ...
    }

    public void stop() {
        ...
    }
}
```

```
        myTi mer . deschedul e(myTi merSpec) ;
        ...
    }

    void ti merWent Off (PTi merWent OffEvent event) {
        PTi merSpec eventTi merSpec = event . getTi merSpec() ;

        // Check whi ch ti mer has caused thi s event; may want
        // to perform di fferent acti ons wi th di fferent ti mers.
        if (eventTi merSpec == myTi merSpec) {
            // Do somethi ng appropri ate.
            ...
        }
    }
}
```

## ***Explicitly store null in variables that point to objects eligible for garbage collection.***

Explicit storage improves the performance of the garbage collector on some platforms. The following example shows one method that waits for items to be placed in a queue and then processes them when they arrive. The current item is stored in the variable `item`. If you don't explicitly store `null` in this variable when you're finished, it continues to hold a live reference to the object until the next time an item is placed in the queue.

```
public void processQueue(Queue queue) {
    QueueItem item;
    while (true) {
        synchronized (queue) {
            while (queue.isEmpty()) {
                try {
                    queue.wait();
                } catch (InterruptedException ex) {
                    return;
                }
            }
            item = queue.removeFirst();
        }
        processItem(item);
        // explicitly store null
        item = null;
    }
}
```

The next example offers one method of removing an item from the queue. The queue in this example is implemented as a linked list with a 'next' field in each

object. When the program removes an item from the queue, it explicitly breaks the reference to the next object in that queue. If it did not explicitly store `null` here, garbage collection could not dispose of the next object in the queue while the current object was still alive.

```
public QueueItem removeFirst() {
    QueueItem ret = head;
    head = head.next;

    // explicitly store null
    ret.next = null;

    return ret;
}
```

### ***Call disposal functions explicitly.***

Call disposal functions (such as `Image.flush` or `Graphics.dispose`) on large external objects when you are done with them. In the following example, the applet loads an image, displays it, and then disposes of it. If it didn't include the two lines in the `destroy` method, the image eventually would be garbage-collected and the resources reclaimed. However, calling methods like `Image.flush` or `Graphics.dispose` immediately helps ensure that enough memory is available when the next applet loads. The following code uses `Image.flush`.

```
import java.awt.*;
import java.applet.*;

public class ImageFlush extends Applet {

    image image;
    public void init() {
        image = getImage(getCodeBase(), "image.gif");
    }

    public void paint(Graphics g) {
        g.drawImage(image, 0, 0, this);
    }
    public void destroy() {
        image.flush();
        image = null;
    }
}
```

## ***Allocate commonly-used resources at applet initialization and hold onto them.***

If an allocation fails, you will know at initialization time that the system doesn't have enough resources for your applet. It's more difficult to recover from a resource problem after you've started performing operations for the user.

## ***Initialize hashtables to an appropriate size.***

When you create hashtables (via `java.util.Hashtable`), specify an appropriate initial size, which should be a prime number. The default size is 101, which often wastes considerable space.

## ***Trim the size of your TextAreas.***

When using a `TextArea` to hold an increasing amount of text (such as output logs or message areas), remember to trim the size occasionally via `replaceRange()` so that it does not grow too large.

The following example inserts two log entries every second into a `TextArea`. It attempts to keep the log length between 400 and 500 bytes. When trimming the log, it removes only whole lines.

```
import java.awt.*;
import java.applet.*;
public class TextTrim extends Applet implements
Runnable {
    TextArea ta;
    Thread logEntry;
    public void init() {
        ta = new TextArea(12, 50);
        ta.setEditable(false);
        add(ta);
    }
    public void start() {
        logEntry = new Thread(this);
        logEntry.start();
    }
    public void stop() {
        logEntry = null;
    }
    public void run() {
        int line = 1;
        while (logEntry == Thread.currentThread()) {
            try {
                Thread.sleep(500);
            } catch (InterruptedException ex) {
```

```
        return;
    }
    ta.append("Log entry: line " + line
+ "\n")
    line++;
    trimLog();
}
// When larger than 500 bytes, try to remove
// at least 100 bytes
public void trimLog() {
    String text = ta.getText();
    if (text.length() > 500) {
        // try to remove only whole lines
        int pos = text.indexOf("\n", 100);
        if (pos != -1) {
            ta.replaceRange("", 0, pos + 1);
        }
    }
}
}
```

### ***Don't rely on finalizers being executed.***

The Java Virtual Machine Specification does not require finalizers to run for applet classes. Virtual machines may delay execution of finalizers arbitrarily and are under no obligation to run finalizers when the virtual machine exits (unless instructed otherwise by `Runtime.runFinalizersOnExit`, which applets may not call). If you have cleanup to do, it is best to do it immediately and explicitly; see “Call disposal functions explicitly.” on page 28.

## Index

### A

- acceptance (of consumer electronics products) 6
- Act i onI nput P r e f e r r e d 14
- animation 24
- applet size 18
- attention (capturing the users') 8

### B

- background color 23
- buffering 22, 24
- But t on. s e t L a b e l 19

### C

- character sets 23
- cl i c k C o u n t 17
- color
  - background 23
  - differences 19
  - text 23
- Component (the class) 25
- consumer acceptance 6
- containers and text 19
- continuous feedback 7
- creat e I m a g e 21

### D

- dialogs
  - modal 11

- display area (of TV) 23
- disposal function 28
- double buffering 24, 25
- double-clicking 10
- downloads 8
- drag and drop 9

### E

- Event . cl i c k C o u n t 17
- Event . m o d i f i e r s 17, 18
- example 16, 28
  - of Act i onI nput P r e f e r r e d 15
  - of double buffering 24
  - of KeyBoar dI nput P r e f e r r e d 14
  - of NoI nput P r e f e r r e d 14
  - of proper implementation of pai nt 21
  - of size control 20
  - of storing null 27
  - of Ti m e r 26
  - of trimming text 29

### F

- feedback 7
- file selection boxes 12
- finalizers 30
- font metrics 19
- full color backgrounds 23

### G

- garbage collection 27, 28
- get G r a p h i c s 20
- get M i n i m u m S i z e 19
- get S i z e 19
- glyphs 23
- go operation 13



## H

heap space 24  
HEIGHT 18  
hierarchical menus 9

## I

I/O 12  
immediate feedback 7  
input devices 13  
input preference interfaces 14  
input/output 12  
invalid date 19  
isDoubleBuffered 24, 25  
ISO Latin 23

## K

KEY\_ACTION 18  
KEY\_PRESS 18  
KEY\_RELEASE 18  
keyboard  
    events 18  
KeyboardInputPreferred 14  
keyboards 13, 14, 16

## L

Latin 1 23

## M

memory management 18, 27, 29  
menus 9, 10, 12  
modal dialogs 11  
motivation (of consumers) 7  
mouse 10, 17  
    events 16, 17  
MOUSE\_DOWN 17  
MOUSE\_UP 17

mouseless environments 13

## N

navigation 8  
NoInputPreferred 14

## O

off-screen 24  
    buffering 25  
    image 21, 24  
on-screen button 15  
overlapping windows 8, 19

## P

paint 20  
PersonalAWT 5  
pixels 23  
popup menus 10  
PositionalInputPreferred 14  
preference API's 14

## R

reflection (test for double buffering) 25  
repair 21, 22  
resource limitations 24

## S

screen  
    distortion 23  
    off-screen buffering 25  
    off-screen image 21  
    on-screen button 15  
    updates 24  
screenphones 18  
screens 10, 13, 15, 22  
scrollbars 22



- scrolling 8
- selection boxes 13
- selection cursor 13
- simplicity 7
- simulated input devices 17
- single-click interaction model 7
- single-pixel lines 23
- size
  - of applets in pixels 18
  - of containers vs. size of text 19
- small typefaces 18
  
- T
- text and containers 19
- text color 23
- Text Area 29
- Thread.sleep 22
- threads 24
- Timer 26
- timer 26
- timer (PersonalJava API) 26
  
- touch screens 10
- TV (screen) 22
- typeface 18, 23
  
- U
- unsupported interface elements 12
- update 20
- URL-based images 24
- user access 13
  
- V
- validate() 19
- virtual keyboard 13
  
- W
- WIDTH 18
- windows 20
  - overlapping 8, 12
  
- X
- XOR-mode 19