PersonalJava™ 1.0 Draft Specification

DRAFT - Version 0.9.2 - DRAFT

Sun Microsystems, Inc.

The attached draft specification of the Personal Java Application Programming Interface (the "Specification") is made available for your review and comment. The Specification will be posted on this web site for at least 60 days. During that time your comments are encouraged. During this period the specification may be revised and reposted in new draft versions. Please submit any comments to the following email address:

personaljava-comments@java.sun.com

Due to the tremendous interest in PersonalJava, Sun may not be able to respond to each submission received on the personaljava-comments@java.sun.com alias.

Sun reserves the right to include or exclude any comments received during this licensee review. Upon completion of the review process we will post the specification on the JavaSoft public web site on a non-confidential basis.

You agree that any comments received from you by Sun during this review will be free for Sun to include in the final published version of the Specification on a non-confidential basis.

For further information on Intellectual Property matters contact Sun Legal Department:

- Trademarks, Jan O'Dell at 415-786-8191
- Patents at 415-336-0069

Introduction

This document provides the specification for PersonalJava 1.0. PersonalJava is a Java™ API and Java Application Environment for networked applications running on personal consumer devices. Because PersonalJava is targeted for platforms such as set-top boxes and smart phones rather than for desktop computers, the PersonalJava API is much smaller than the JDK 1.1 API. This makes PersonalJava highly scalable and configurable while using minimal memory.

This document is intended as a guide to the facilities Personal Java provides to applications, applet, and their developers.

PersonalJava is designed to meet the following goals:

- Applets written to the PersonalJava specification should run in a JDK 1.1 applet environment.
- PersonalJava applets should be link-compatible with the JDK 1.1 packages that PersonalJava supports. Authors should be able to write applets for PersonalJava that can use JDK 1.1 applet features when running in a JDK 1.1 applet environment.
- Products that are based on PersonalJava should be usable by people with no computer experience.
- PersonalJava should provide flexibility for dealing with the many input and output formats found in the consumer electronics market, such as remote controls, television output, and touch screens.
- PersonalJava should require less memory than JDK 1.1, in terms of the Java system itself and of runtime requirements. We intend that the Personal Java virtual machine and class libraries fit comfortably in 2 megabytes of ROM, and execute in 1-2 megabytes of RAM.
- PersonalJava should run a large proportion of the applets written for JDK 1.0.2 and JDK 1.1.

Items still under investigation/work in progress may be found here.

Changes to the specification since the previous version may be found here.

The PersonalJava 1.0 API

The PersonalJava 1.0 API is a subset of the JDK 1.1 API, supplemented by a small number of new APIs designed to meet the needs of networked embedded applications. Java APIs introduced after JDK 1.1 will not automatically become a part of the PersonalJava API. New APIs will be reviewed and evaluated for appropriateness before being added to PersonalJava.

The JDK 1.1 packages included in PersonalJava are:

- `java.applet`
- `java.awt`
- `java.awt.datatransfer`
- `java.awt.event`
- `java.awt.image`
- `java.awt.peer`
- `java.beans`
- `java.io`
- `java.lang`
- `java.lang.reflect`
- `java.net`
- `java.util`
- `java.util.zip`

These packages provide the core capabilities of Java, as reflected in the many books on Java programming, as observed in our samples of current applet and application usage, and as reflects the capabilities of "small" devices and their users (insofar as it is possible to make generalizations about either).

Some of the functionality provided by the `java.awt`, `java.io`, and `java.util` packages is not appropriate for inclusion in PersonalJava, however. Examples of such inappropriate functionality are top-level windows and some features of internationalization. PersonalJava does not require support for these features. Unsupported and optional features are detailed below and in the documentation for the individual packages.

Definitions:

- An **unsupported class** is one that has all of its methods present (which is required for link-compatibility), but these methods will typically throw `java.lang.UnsupportedOperationException`, as documented.
- An **unsupported method** is one that is present, has the same signature as its equivalent in the full JDK, but which usually throws `java.lang.UnsupportedOperationException`, as documented.
- An **optional class** is one that is not required to be present, and which applets should not assume to be present, but which particular implementations might include to provide additional capabilities to built-in applications or special downloaded applets.
- An **optional package** is a Java package consisting entirely of **optional classes.**

The following table lists the classes that are optional in PersonalJava along with their dependencies. The classes are grouped in subsets that must be implemented together.

Package/Class	Dependencies
`java.math`	
`java.security` `java.security.interfaces`	`Mutual, plus java.math`

`java.awt.CheckboxMenuItem`
`java.awt.Dialog (non-modal)`
`java.awt.Frame`
`java.awt.Menu`
`java.awt.MenuBar`
`java.awt.MenuShortcut`
`java.awt.Window`
	Mutual; if implemented, must be as a group.

`java.awt.FileDialog`	
`java.awt.PopupMenu (nested)`	
`java.net.ServerSocket`	
`java.util (unsupported classes)`	
`java.util.zip (unsupported classes)`	

Note that `java.util` and `java.util.zip` "unsupported classes" function as optional classes because they may be implemented as long as they conform to JDK 1.1 standards. Unlike the `java.awt` classes listed above, they are not grouped into subsets that must be implemented together.

The JDK 1.1 packages not included in the PersonalJava API are:

- `java.rmi`
- `java.rmi.dgc`
- `java.rmi.registry`
- `java.rmi.server`
- `java.security.acl`
- `java.sql`
- `java.text`
- `java.text.resources`

These packages provide capabilities that are more likely to be required by "enterprise", applications than small consumer devices, which are not yet widely used, which require amounts of ROM beyond what most consumer devices are able to accomodate, or some or all of the above.

The remainder of this specification consists of descriptions of the packages contained in the PersonalJava API. The description for each package contains a table that lists the classes and interfaces for that package. In the tables, classes are marked as *required*, *modified*, *unsupported*, or *optional*. Required classes are fully implemented as in the JDK 1.1 specification. A class is marked as modified if any of its methods are not supported or are modified in the PersonalJava API. Descriptions for any optional or modified methods are provided following the class and interface tables for each package.

If a class is marked as unsupported, calling its constructor or calling any of its static methods will result in a `java.lang.UnsupportedOperationException`. Any instances of unsupported classes that are normally held in static class variables will instead hold `null`.

Unsupported methods will throw a `java.lang.UnsupportedOperationException` when called.

PersonalJava has added new APIs to the `java.awt`, `java.lang`, and `java.util` packages. These additional APIs are described in the documentation for those packages.

java.applet

PersonalJava supports the full JDK 1.1 API for the `java.applet` package.

Class and Interface List

Name	Status
`Applet`	Required
`AppletContext`	Required
`AppletStub`	Required
`AudioClip`	Required

The `APPLET` HTML tag

The `APPLET` tag embeds a PersonalJava applet within a Web page. It conforms to the following SGML Document Type Definition (DTD):

```
<!ENTITY % Length "CDATA" -- nn for pixels or nn% for percentage length -->
<!ENTITY % Pixels "CDATA" -- integer representing length in pixels -->
<!ENTITY % IAlign "(top|middle|bottom|left|right)" -- alignment -->
<!ENTITY % URL    "CDATA" -- a Uniform Resource Locator -->

<!ELEMENT APPLET - - (%text)* +(PARAM)>

<!ATTLIST APPLET
        codebase %URL     #IMPLIED   -- code base --
        code     CDATA    #IMPLIED   -- class file --
        alt      CDATA    #IMPLIED   -- text for display in place of applet --
        name     CDATA    #IMPLIED   -- applet name --
        archive  CDATA    #IMPLIED   -- archive attribute for JAR files --
        object   CDATA    #IMPLIED   -- name of the file containing a
                                        serialized representation of an
                                        applet, which will be deserialized.
                                        One of CODE or OBJECT must be
                                        present.  See the JDK 1.1 APPLET tag
                                        documentation for details. --
        width    %Length  #REQUIRED  -- suggested width in pixels or as percenta
        height   %Length  #REQUIRED  -- suggested height in pixels or as percent
        align    %IAlign  baseline   -- vertical or horizontal alignment --
        hspace   %Pixels  #IMPLIED   -- suggested horizontal gutter --
        vspace   %Pixels  #IMPLIED   -- suggested vertical gutter --
        >
```

The `WIDTH` and `HEIGHT` attributes of the applet tag provide only rough indications of the applet's dimensions. The display area of consumer devices is often smaller than that of desktop computer systems. An applet viewer or browser may need to shrink the applet to enable it to fit in a device's display area.

java.awt

PersonalJava does not require that a platform implement the full functionality of the JDK 1.1 `java.awt` package. The following is a description of the subset of the JDK 1.1 `java.awt` package that is the baseline specification for the PersonalJava AWT functionality. Licensees may implement additional features of the JDK 1.1 `java.awt` package as needed (that is, implement those items marked as *optional* or *unsupported* below), provided that the additional features conform fully to the JDK 1.1 `java.awt` package specification.

Further, implementing some of the features may require implementing others, to provide consistency. The dependencies are explained below.

PersonalJava also provides new APIs for AWT functionality not found in the JDK 1.1 API.

Class and Interface List

Name	Required	Optional	Modified	Unsupported
`AWTError`	✔			
`AWTEvent`	✔			
`AWTEventMulticaster`	✔			
`AWTException`	✔			
`Adjustable`	✔			
`BorderLayout`	✔			
`Button`	✔			
`Canvas`	✔			
`CardLayout`	✔			
`Checkbox`	✔			
`CheckboxGroup`	✔			
`CheckboxMenuItem`		✔		
`Choice`	✔			
`Color`	✔			
`Component`			✔	
`Container`	✔			

`Cursor`	✔			
`Dialog`			✔	
`Dimension`	✔			
`Event`	✔			
`EventQueue`	✔			
`FileDialog`		✔		
`FlowLayout`	✔			
`Font`	✔			
`FontMetrics`	✔			
`Frame`		✔		
`Graphics`			✔	
`GridBagConstraints`	✔			
`GridBagLayout`	✔			
`GridLayout`	✔			
`IllegalComponentStateException`	✔			
`Image`	✔			
`Insets`	✔			
`ItemSelectable`	✔			
`Label`	✔			
`LayoutManager`	✔			
`LayoutManager2`	✔			
`List` (see Cautions)	✔			
`MediaTracker`	✔			
`Menu`		✔		
`MenuBar`		✔		
`MenuComponent`	✔			
`MenuContainer`	✔			
`MenuItem`	✔			
`MenuShortcut`		✔		
`Panel`	✔			
`Point`	✔			

`Polygon`	✔			
`PopupMenu`			✔	
`PrintGraphics (see Toolkit)`	✔			
`PrintJob (see Toolkit)`	✔			
`Rectangle`	✔			
`ScrollPane`			✔	
`Scrollbar`				✔
`Shape`	✔			
`SystemColor`	✔			
`TextArea`			✔	
`TextComponent`	✔			
`TextField`	✔			
`Toolkit`			✔	
`Window`		✔		

Modified, Optional, and Unsupported Class Details

Method	Status	Comments
`CheckboxMenuItem()`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if the `Menu` constructor does not throw `java.lang.UnsupportedOperationException`, this one may not either, and vice-versa.
`CheckboxMenuItem(String)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if the `Menu` constructor does not throw `java.lang.UnsupportedOperationException`, this one may not either, and vice-versa.

`CheckboxMenuItem(String, boolean)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if the `Menu` constructor does not throw `java.lang.UnsupportedOperationException`, this one may not either, and vice-versa.
`Component.setCursor(Cursor)`	Modified	The specified cursor may be ignored. Some platforms may not support cursors, while others may limit the types of cursors displayed for usability reasons.
`Dialog(Frame)`	Modified	This constructor normally creates a modeless `Dialog`. However, if a given implementation of PersonalJava does not allow the creation of `Frame`s, it cannot allow the creation of modeless `Dialog`s, and vice-versa. In that case, this constructor will throw `java.lang.UnsupportedOperationException` if called.
`Dialog(Frame, boolean)`	Modified	This constructor can create a modal or modeless `Dialog`, subject to the value of its `boolean` argument. There are restrictions on the circumstances under which modeless `Dialog`s may be created; see above for those. An implementation is permitted to allow only one modal dialog to be visible at a time; in this case, if an applet tries to display a dialog when one is already visible, the visible dialog may be hidden. However, when the new modal dialog disappears, the original dialog should become visible again.
`Dialog(Frame, String)`	Modified	This constructor creates a modeless `Dialog`; see above for a description of the restrictions PersonalJava imposes.
`Dialog(Frame, String, boolean)`	Modified	PersonalJava imposes restrictions on modeless `Dialog`s; see above. An implementation is permitted to allow only one modal dialog to be visible at a time; in this case, if an applet tries to display a dialog when one is already visible, the visible dialog may be hidden. However, when the new modal dialog disappears, the original dialog should become visible again. The `String` passed as the title may be ignored.
`Dialog.setResizable(boolean)`	Modified	The specified value may be ignored.

`FileDialog(Frame)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. Assuming normal applet security restrictions are in force, applets are forbidden to access the local file system, making `FileDialog` useless to them. If a given implementation includes a filesystem that is intended to be visible to the user, this constructor must not throw `java.lang.UnsupportedOperationException`, and should operate normally.
`FileDialog(Frame, String)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. (See note above.)
`FileDialog(Frame, String, int)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. (See note above.)
`Frame()`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if a given implementation gives the user control over the placement of overlapping windows (`Frame`s), this constructor must operate normally. (See also the description of class `Window`.) Note that every implementation of PersonalJava must provide a `Frame` instance as the root of any component hierarchy; programs will need to use this instance as an argument to the several AWT methods and constructors that expect one.
`Frame(String)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. (See note above.)
`Graphics.setXORMode()`	Modified	Some displays, notably anti-aliased ones, are not capable of drawing in exclusive-or mode. Implementations in which this is the case will throw `java.lang.UnsupportedOperationException` when this method is called.

`Menu()`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if a given platform implements the `Frame` constructor such that it does not throw `java.lang.UnsupportedOperationException`, it must implement `Menu` likewise, and vice-versa.
`Menu(String)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. (See note above.)
`Menu(String, boolean)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. (See note above.)
`MenuBar()`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if a given platform implements the `Frame` constructor such that it does not throw `java.lang.UnsupportedOperationException`, it must implement `MenuBar` likewise, and vice-versa.
`MenuShortcut(int)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if a given platform implements the `MenuBar` constructor such that it does not throw `java.lang.UnsupportedOperationException`, it must implement `MenuShortcut` likewise, and vice-versa.
`MenuShortcut(int, boolean)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. (See note above.)
`PopupMenu.add(MenuItem)`	Modified	If an argument other than an instance of `MenuItem`, *excluding* subclasses, is passed to this method, a `UnsupportedOperationException` may result when `PopupMenu.show` is called.
`Scrollbar()`	Unsupported	This constructor is not supported, and will throw `java.lang.UnsupportedOperationException` if called.
`Scrollbar(int)`	Unsupported	This constructor is not supported, and will throw `java.lang.UnsupportedOperationException` if called.

`Scrollbar(int, int, int, int, int)`	Unsupported	This constructor is not supported, and will throw `java.lang.UnsupportedOperationException` if called.
`ScrollPane()`	Modified	The scrollbar display policy may be interpreted specially. A platform may substitute other scrolling mechanisms in place of scrollbars. See note.
`ScrollPane(int)`	Modified	The scrollbar display policy may be ignored. A platform may substitute other scrolling mechanisms in place of scrollbars. See note.
`TextArea()`	Modified	The scrollbar display policy may be ignored. A platform may substitute other scrolling mechanisms in place of scrollbars. See note.
`TextArea(String)`	Modified	The scrollbar display policy may be ignored. A platform may substitute other scrolling mechanisms in place of scrollbars. See note.
`TextArea(int, int)`	Modified	The scrollbar display policy may be ignored. A platform may substitute other scrolling mechanisms in place of scrollbars. See note.
`TextArea(String, int, int)`	Modified	The scrollbar display policy may be ignored. A platform may substitute other scrolling mechanisms in place of scrollbars. See note.
`TextArea(String, int, int, int)`	Modified	The scrollbar display policy may be ignored. A platform may substitute other scrolling mechanisms in place of scrollbars. See note.
`Toolkit.getPrintJob(Frame, String, Properties)`	Modified	A program prints by first calling this method to obtain a `PrintJob` object, from which it obtains a series of objects implementing the `PrintGraphics` interface. If a given implementation omits printer support, `getPrintJob` should throw `java.lang.UnsupportedOperationException`.
`Window(Frame)`	Optional	This constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. However, if the `Frame` constructor does not throw `java.lang.UnsupportedOperationException`, this one may not either, and vice-versa.

Cautions

A platform's implementation of `List` may provide a scrolling mechanism other than scrollbars. See note.

Implementation of Scrolling Controls and Behavior

Because scrolling in Personal Java can be done using whatever means is appropriate for the system, rather than being restricted to traditional computer desktop metaphors, the scrollbar display policies (e.g. for `ScrollPane`) have slightly different meanings:

`ScrollPane` scrollbar policy	**Visual Effect (look)**	**Functional Effect (feel)**
`SCROLLBARS_NEVER`	Users see no indication that the item supports scrolling.	Scrolling this item can only be done programmatically.
`SCROLLBARS_AS_NEEDED`	If visual feedback for scrolling is supported, it should be given only if the size of the scrollable area requires it.	Scrolling can be done by the user using whatever means the toolkit provides.
`SCROLLBARS_ALWAYS`	If visual feedback for scrolling is supported, it should be given even if the size of the scrollable area does not require it.	Scrolling can be done by the user using whatever means the toolkit provides.

Similarly, the scroller display policy for `TextArea` is also slightly modified:

`TextArea` scrollbar policy	Visual Effect (look)	Functional Effect (feel)
SCROLLBARS_NONE	Users see no indication that the `TextArea` supports scrolling.	Scrolling this `TextArea` can only be done programmatically.
SCROLLBARS_VERTICAL_ONLY	If visual feedback for scrolling is supported, it should be given only if the height of the scrollable area requires it.	Scrolling can be done by the user using whatever means the toolkit provides.
SCROLLBARS_HORIZONTAL_ONLY	If visual feedback for scrolling is supported, it should be given only if the width of the scrollable area requires it.	Scrolling can be done by the user using whatever means the toolkit provides.
SCROLLBARS_BOTH	If visual feedback for scrolling is supported, it should be given for the horizontal and vertical axes.	Scrolling can be done by the user using whatever means the toolkit provides.

APIs for Double Buffering and Specifying Component Behavior

PersonalJava provides additional APIs for performing double buffering and for specifying ways of interfacing with components in a mouseless environment. (Ideally, these APIs will be folded into a future release of the JDK.)

Methods for Double Buffering

PersonalJava provides a new method, `isDoubleBuffered`, for class `Component`.

```
// in class java.awt.Component
public boolean isDoubleBuffered();
```

> If `isDoubleBuffered` returns true, then all drawing done inside `paint` and `update` methods will be double buffered automatically. The default value for the double buffering setting is platform-specific.
>
> To achieve double buffering on a platform that does not support it, developers should 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.

Developers wishing to use these methods in other versions of the JDK should use the compatibility interface provided in `sunw.awt`:

```
package sunw.awt;
public interface DoubleBuffering {
    public boolean isDoubleBuffered();
}
```

These interfaces will be implemented by the `Component` class in the PersonalJava API.

Example: Using the Double Buffering Interface

• Applet class `MyAnimator` checks at `init` time to see if the hardware will automatically double-buffer its graphical output. If not, it allocates its own off-screen image buffer:

```
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...
        if (! isDoubleBuffered()) {
            allocateOffScreen();
        }
        ...
    }

    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.
            // First draw offscreen 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);
        }
    }

    ...
}
```

Interfaces for Specifying Component Behavior

The PersonalJava API includes four interfaces that allow developers to make it easier for their applets and applications to adapt to mouseless environments. Examples of such environments are keyboard-only systems and systems that are operated by remote control.

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 will provide a way for the user to "select" the current component, indicating that the user desires to interact with the component. For example, after navigating to an on-screen button component, the user might press the `Go` key on a remote control to indicate that the on-screen button is to be "pressed".

PersonalJava provides input preference interfaces to allow developers to specify the manner in which users navigate among components and the way that users interact with components. These interfaces and their descriptions are:

```
package java.awt;
public interface NoInputPreferred {}
public interface KeyboardInputPreferred {}
public interface ActionInputPreferred {}
public interface PositionalInputPreferred {}
```

`NoInputPreferred`

> The user may not navigate to the component. This interface might be appropriate for components such as labels.

`KeyboardInputPreferred`

> The component will be used primarily via keyboard input. Some platforms will respond by popping up an on-screen keyboard when the user navigates to or selects this component. The developer should ensure that `Component.isFocusTraversable` returns `true` for this component.

`ActionInputPreferred`

This interface is intended for those types of components that the user would click on or otherwise "activate" using the input device. The component should receive a `MOUSE_ENTER` event when the user navigates to the component, and a `MOUSE_EXIT` event when the user navigates from the component. It should receive a `MOUSE_DOWN` event followed by a `MOUSE_UP` event when the user selects it. The mouse coordinates for all events associated with this component will be the coordinates of the center of the component.

`PositionalInputPreferred`

The component will be used primarily by the user selecting x,y coordinates within the component. The component should receive `MOUSE_ENTER` and `MOUSE_EXIT` events when the user navigates to it. The system should provide some mechanism for selecting specific x,y coordinates within the component's bounds, and provide mouse movement events if possible. The platform should decide if selection of x,y coordinate begins when the user navigates to the component or when the user selects the component. In either case, the component should receive a `MOUSE_DOWN` and `MOUSE_UP` event with the selected coordinates.

Examples: Using the Component Behavior Interfaces

- Class `MyLabel` represents an area that simply displays a test string. It isn't concerned with any sort of input at all:

```
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 into which the user can type a number. It only concerns itself with character input, but not positional or other sorts of input:

```
public class MyNumberField extends Canvas implements KeyboardInputPreferred, KeyList

    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 we make it implement `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 on which the user can "draw." It concerns itself with "mouse" up and down events, keyboard input, and the position of the input device. Note that because we wish this Component to act more like a drawing pad than a type-in field, we have it implement `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 user is pressing the shift key, draw a line, else
       just move. */
    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;
    }

}
```

A component should implement only one of the input preference interfaces. If a component implements more than one, it should behave as if none had been implemented.

Input preferences are meant as hints to the implementation, and some implementations may ignore them on certain Components. For example, an implementation might choose always to treat a `Button` (or a subclass of `Button`) as implementing `ActionInputPreferred`, irrespective of the input preference interfaces actually implemented. However, implementations should honor input preferences on subclasses of the generic `Canvas`, `Component`, `Container`, `Dialog`, `Frame`, `ScrollPane`, and `Window` classes when appropriate. All classes in the `java.awt` package are guaranteed not to implement any of these interfaces.

An implementation should ignore these input preferences when appropriate. For example, on a platform with a keyboard and mouse, the implementation might always ignore them. On a platform with a remote control that contains a trackball, the implementation might ignore all but `KeyboardInputPreferred`. On a platform with only a keyboard, the implementation might ignore only `KeyboardInputPreferred`.

Note that a component that inherits from `NoInputPreferred` may still receive events on an implementation that chooses to ignore it. Also, an implementation may choose to not allow the user to navigate to a component for reasons not covered here. For example, an implementation might not allow a user to navigate to a component that is too small or that is obscured.

If a component does not inherit from any of these interfaces, an implementation might establish a default for predictability, might attempt to detect what the component wants based on event listeners, or might apply other heuristics.

Developers wishing to use these features under other versions of the JDK should use the compatibility interfaces in `sunw.awt`:

```
package sunw.awt;
public interface NoInputPreferred {}
public interface KeyboardInputPreferred {}
public interface ActionInputPreferred {}
public interface PositionalInputPreferred {}
```

java.awt.datatransfer

PersonalJava supports the full JDK 1.1 API for the `java.awt.datatransfer` package, which is invaluable for transfer of information both between Java applications and/or applets and between Java and non-Java programs running on the same platform.

Class and Interface List

Name	Status
`Clipboard`	Required
`ClipboardOwner`	Required
`DataFlavor`	Required
`StringSelection`	Required
`Transferable`	Required
`UnsupportedFlavorException`	Required

java.awt.event

PersonalJava supports the full JDK 1.1 API for the `java.awt.event` package.

Class and Interface List

Name	Status
ActionEvent	Required
ActionListener	Required
AdjustmentEvent	Required
AdjustmentListener	Required
ComponentAdapter	Required
ComponentEvent	Required
ComponentListener	Required
ContainerAdapter	Required
ContainerEvent	Required
ContainerListener	Required
FocusAdapter	Required
FocusEvent	Required
FocusListener	Required
InputEvent	Required
ItemEvent	Required
ItemListener	Required
KeyAdapter	Required
KeyEvent	Required
KeyListener	Required
MouseAdapter	Required
MouseEvent	Required
MouseListener	Required
MouseMotionAdapter	Required
MouseMotionListener	Required
PaintEvent	Required
TextEvent	Required
TextListener	Required
WindowAdapter	Required
WindowEvent	Required

`WindowListener`	Required

java.awt.image

PersonalJava supports the full JDK 1.1 API for the `java.awt.image` package.

Class and Interface List

Name	Status
`AreaAveragingScaleFilter`	Required
`ColorModel`	Required
`CropImageFilter`	Required
`DirectColorModel`	Required
`FilteredImageSource`	Required
`ImageConsumer`	Required
`ImageFilter`	Required
`ImageObserver`	Required
`ImageProducer`	Required
`IndexColorModel`	Required
`MemoryImageSource`	Required
`PixelGrabber`	Required
`RGBImageFilter`	Required
`ReplicateScaleFilter`	Required

java.awt.peer

Developers should not directly use the interfaces in `java.awt.peer`, unless they are porting PersonalJava to a new platform. An implementation should mirror the restrictions from `java.awt` in an implementation of peers for PersonalJava. For example, an implementation of `PopupMenuPeer.show()` for PersonalJava may throw `UnsupportedOperationException` when called, assuming `PopupMenu.show()` does.

java.beans

PersonalJava supports the full JDK 1.1 API for the `java.beans` package.

Class and Interface List

Name	Status
`BeanDescriptor`	Required
`BeanInfo`	Required
`Beans`	Required
`Customizer`	Required
`EventSetDescriptor`	Required
`FeatureDescriptor`	Required
`IndexedPropertyDescriptor`	Required
`IntrospectionException`	Required
`Introspector`	Required
`MethodDescriptor`	Required
`ParameterDescriptor`	Required
`PropertyChangeEvent`	Required
`PropertyChangeListener`	Required
`PropertyChangeSupport`	Required
`PropertyDescriptor`	Required
`PropertyEditor`	Required
`PropertyEditorManager`	Required
`PropertyEditorSupport`	Required
`PropertyVetoException`	Required
`SimpleBeanInfo`	Required
`VetoableChangeListener`	Required
`VetoableChangeSupport`	Required
`Visibility`	Required

java.io

Class and Interface List

Name	Status
`BufferedInputStream`	Required
`BufferedOutputStream`	Required
`BufferedReader`	Required
`BufferedWriter`	Required
`ByteArrayInputStream`	Required
`ByteArrayOutputStream (see Cautions)`	Required
`CharArrayReader`	Required
`CharArrayWriter`	Required
`CharConversionException`	Required
`DataInput`	Required
`DataInputStream`	Required
`DataOutput`	Required
`DataOutputStream`	Required
`EOFException`	Required
`Externalizable`	Required
`File`	Required
`FileDescriptor`	Required
`FileInputStream`	Required
`FileNotFoundException`	Required
`FileOutputStream`	Required
`FileReader`	Required
`FileWriter`	Required
`FilenameFilter`	Required
`FilterInputStream`	Required
`FilterOutputStream`	Required
`FilterReader`	Required
`FilterWriter`	Required
`IOException`	Required

InputStream	Required
InputStreamReader (see Cautions)	Required
InterruptedIOException	Required
InvalidClassException	Required
InvalidObjectException	Required
LineNumberInputStream	Required
LineNumberReader	Required
NotActiveException	Required
NotSerializableException	Required
ObjectInput	Required
ObjectInputStream	Required
ObjectInputValidation	Required
ObjectOutput	Required
ObjectOutputStream	Required
ObjectStreamClass	Required
ObjectStreamException	Required
OptionalDataException	Required
OutputStream	Required
OutputStreamWriter (see Cautions)	Required
PipedInputStream	Required
PipedOutputStream	Required
PipedReader	Required
PipedWriter	Required
PrintStream	Required
PrintWriter	Required
PushbackInputStream	Required
PushbackReader	Required
RandomAccessFile	Required
Reader	Required
SequenceInputStream	Required
Serializable	Required

`StreamCorruptedException`	Required
`StreamTokenizer`	Required
`StringBufferInputStream`	Required
`StringReader`	Required
`StringWriter`	Required
`SyncFailedException`	Required
`UTFDataFormatException`	Required
`UnsupportedEncodingException`	Required
`WriteAbortedException`	Required
`Writer`	Required

Cautions

Developers should be aware that the specific character encodings available (as used in `ByteArrayOutputStream`, `InputStreamReader`, and `OutputStreamWriter`) are platform specific and may be quite limited in PersonalJava implementations. However, an implementation should guarantee the availability of converters for ISO 8859-1 ("Latin-1"), Unicode (big- and little-endian varieties, both marked and unmarked), and the native character encoding of the platform itself.

java.lang

PersonalJava supports the full JDK 1.1 API for the `java.lang` package. In addition, PersonalJava provides an additional API for a `UnsupportedOperationException` class.

Class and Interface List

Name	Status
`AbstractMethodError`	Required
`ArithmeticException`	Required
`ArrayIndexOutOfBoundsException`	Required
`ArrayStoreException`	Required
`Boolean`	Required
`Byte`	Required
`Character`	Required
`Class`	Required

ClassCastException	Required
ClassCircularityError	Required
ClassFormatError	Required
ClassLoader	Required
ClassNotFoundException	Required
CloneNotSupportedException	Required
Cloneable	Required
Compiler	Required
Double	Required
Error	Required
Exception	Required
ExceptionInInitializerError	Required
Float	Required
IllegalAccessError	Required
IllegalAccessException	Required
IllegalArgumentException	Required
IllegalMonitorStateException	Required
IllegalStateException	Required
IllegalThreadStateException	Required
IncompatibleClassChangeError	Required
IndexOutOfBoundsException	Required
InstantiationError	Required
InstantiationException	Required
Integer	Required
InternalError	Required
InterruptedException	Required
LinkageError	Required
Long	Required
Math	Required
NegativeArraySizeException	Required
NoClassDefFoundError	Required

`NoSuchFieldError`	Required
`NoSuchFieldException`	Required
`NoSuchMethodError`	Required
`NoSuchMethodException`	Required
`NullPointerException`	Required
`Number`	Required
`NumberFormatException`	Required
`Object`	Required
`OutOfMemoryError`	Required
`Process`	Required
`Runnable`	Required
`Runtime`	Required
`RuntimeException`	Required
`SecurityException`	Required
`SecurityManager`	Required
`Short`	Required
`StackOverflowError`	Required
`String (see Cautions)`	Required
`StringBuffer`	Required
`StringIndexOutOfBoundsException`	Required
`System`	Required
`Thread`	Required
`ThreadDeath`	Required
`ThreadGroup`	Required
`Throwable`	Required
`UnknownError`	Required
`UnsatisfiedLinkError`	Required
`VerifyError`	Required
`VirtualMachineError`	Required
`Void`	Required

Cautions

Developers should be aware that the specific character encodings available (as used in `String`) are platform specific and may be quite limited in PersonalJava implementations.

Additional APIs

`UnsupportedOperationException` is defined as:

```
package java.lang;
public class UnsupportedOperationException
        extends sunw.lang.UnsupportedOperationException {
    public UnsupportedOperationException(Object o);
    public UnsupportedOperationException(Object o, String s);
}
```

For compatibility, the following class is also provided and can be included in downloaded applets:

```
package sunw.lang;
public class UnsupportedOperationException extends RuntimeException {
    public UnsupportedOperationException(Object o);
    public UnsupportedOperationException(Object o, String s);
}
```

Developers wishing to use this feature under other versions of the JDK should use the `sunw.lang` version of `UnsupportedOperationException`.

java.lang.reflect

PersonalJava supports the full JDK 1.1 API for the `java.lang.reflect` package.

Class and Interface List

Name	Status
`Array`	Required
`Constructor`	Required
`Field`	Required
`InvocationTargetException`	Required
`Member`	Required
`Method`	Required
`Modifier`	Required

java.net

PersonalJava supports the full JDK 1.1 API for the `java.net` package, excepting optional class `ServerSocket`, which is rendered off-limits for applets by most `java.lang.SecurityManager` implementations.

Class and Interface List

Name	Status
`BindException`	Required
`ConnectException`	Required
`ContentHandler`	Required
`ContentHandlerFactory`	Required
`DatagramPacket`	Required
`DatagramSocket`	Required
`DatagramSocketImpl`	Required
`FileNameMap`	Required
`HttpURLConnection`	Required
`InetAddress`	Required
`MalformedURLException`	Required
`MulticastSocket`	Required
`NoRouteToHostException`	Required
`ProtocolException`	Required
`ServerSocket`	Optional
`Socket`	Required
`SocketException`	Required
`SocketImpl`	Required
`SocketImplFactory`	Required
`URL`	Required
`URLConnection`	Required
`URLEncoder`	Required
`URLStreamHandler`	Required
`URLStreamHandlerFactory`	Required
`UnknownHostException`	Required
`UnknownServiceException`	Required

java.util

PersonalJava does not support the full functionality of the JDK 1.1 `java.util` package. The following is a description of the subset of the JDK 1.1 `java.util` package that is the baseline specification for the PersonalJava utility functionality. Licensees may implement additional functionality from the JDK 1.1 `java.util` package as needed, provided that the additional functionality conforms fully to the JDK 1.1

`java.util` package specification.

PersonalJava also provides new APIs for utility features not found in the JDK 1.1 API.

Class and Interface List

Name	Status
`BitSet`	Required
`Calendar`	Unsupported
`Date`	Modified
`Dictionary`	Required
`EmptyStackException`	Required
`Enumeration`	Required
`EventListener`	Required
`EventObject`	Required
`GregorianCalendar`	Unsupported
`Hashtable`	Required
`ListResourceBundle`	Required
`Locale`	Required
`MissingResourceException`	Required
`NoSuchElementException`	Required
`Observable`	Required
`Observer`	Required
`Properties`	Required
`PropertyResourceBundle`	Required
`Random`	Required
`ResourceBundle`	Required
`SimpleTimeZone`	Unsupported
`Stack`	Required
`StringTokenizer`	Required
`TimeZone`	Unsupported
`TooManyListenersException`	Required
`Vector`	Required

Modified Class Details

Class `Date` and its methods are not deprecated in PersonalJava.

java.util.zip

PersonalJava does not support the full functionality of the JDK 1.1 `java.util.zip` package. The following is a description of the subset of the JDK 1.1 `java.util.zip` package that is the baseline specification for the PersonalJava utility functionality. Licensees may implement additional functionality from the JDK 1.1 `java.util.zip` package as needed, provided that the additional functionality conforms fully to the JDK 1.1 `java.util.zip` package specification.

The PersonalJava `java.util.zip` package is intended to provide at least the capabilities necessary to allow applets and other data to be loaded from JAR files.

Class and Interface Details

Method	Status	Comments
`Adler32`	Unsupported	The Adler32 constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. The same is true of all of the other methods of `Adler32`.
`CRC32`	Required	
`CheckedInputStream`	Required	
`CheckedOutputStream`	Required	
`Checksum`	Required	
`DataFormatException`	Required	
`Deflater`	Unsupported	The Deflater constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. The same is true of all of the other methods of `Deflator`.
`DeflaterOutputStream`	Unsupported	The DeflaterOutputStream constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. The same is true of all of the other methods of `DeflatorOutputStream`.
`GZIPInputStream`	Required	

`GZIPOutputStream`	Unsupported	The GZIPOutputStream constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. The same is true of all of the other methods of `GZIPOutputStream`.
`Inflater`	Modified	The constructor `Inflater()` will throw `UnsupportedOperationException` when called, as the `nowrap` option (implicit in this call) is not supported. Likewise, `Inflater(boolean)` will throw `UnsupportedOperationException` when called with a `false` argument.
`InflaterInputStream`	Required	
`ZipConstants`	Required	
`ZipEntry`	Required	
`ZipException`	Required	
`ZipFile`	Unsupported	The ZipFile constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. The same is true of all of the other methods of `ZipFile`.
`ZipInputStream`	Required	
`ZipOutputStream`	Unsupported	The ZipOutputStream constructor is not required to be implemented, and may throw `java.lang.UnsupportedOperationException` if called. The same is true of all of the other methods of `ZipOutputStream`.

Timer and TimerSpec APIs

PersonalJava also includes the following classes and interfaces in `sunw.util`. (Ideally, to ensure wide compatibility, these APIs will be integrated into a future version of the JDK.)

```
package sunw.util;

public abstract class Timer {

    public static Timer getTimer();

    public abstract void schedule(TimerSpec t);
    public abstract void deschedule(TimerSpec t);
}

public class TimerSpec {

    // defaults to a one-shot timer going off immediately
    // regular defaults to true
    public TimerSpec() {
```

```
        setAbsolute(false);
        setRepeat(false);
        setRegular(true);
        setTime(0);
    }

    // accessors

    public void setAbsolute(boolean absolute);
    public boolean isAbsolute();

    public void setRepeat(boolean repeat);
    public boolean isRepeat();

    // ala regular vs. irregular in sun.misc.Timer
    public void setRegular(boolean regular);
    public boolean isRegular();

    public void setTime(long time);
    public long getTime();

    // listeners

    public void addTimerWentOffListener(TimerWentOffListener l);
    public void removeTimerWentOffListener(TimerWentOffListener l);

    // convenience functions

    public void setAbsoluteTime(long when) {
        setAbsolute(true);
        setTime(when);
        setRepeat(false);
    }

    public void setDelayTime(long delay) {
        setAbsolute(false);
        setTime(delay);
        setRepeat(false);
    }

    // for the benefit of timer implementations

    public void notifyListeners(Timer source);
}

public class TimerWentOffEvent extends java.util.EventObject {

    private TimerSpec timerSpec;

    public TimerWentOffEvent(Timer source, TimerSpec spec) {
        super(source);

        timerSpec = spec;
    }

    public TimerSpec getTimerSpec() {
        return timerSpec;
    }
}

public interface TimerWentOffListener {
```

```
        void timerWentOff(TimerWentOffEvent e);
    }
```

The `Timer` methods are:

`getTimer()`

Returns a timer object. There may be one `Timer` instance per virtual machine, one per applet, one per call to `getTimer`, or some other platform dependent implementation.

`schedule(TimerSpec)`

Begins monitoring a `TimerSpec`. When the timer specification should fire, the timer will call `TimerSpec.notifyListeners`.

`deschedule(TimerSpec)`

Removes a `TimerSpec` from the set of monitored specifications. The descheduling happens as soon as practical, but may not happen immediately.

The `TimerSpec` methods are:

`TimerSpec()`

Creates a new timer specification that defaults to a one-shot regular timer that goes off after a delay of 0ms.

`setAbsolute(boolean)`
`isAbsolute()`

Determines whether the time specification in `setTime` is an absolute time or a delay time.

`setRepeat(boolean)`
`isRepeat()`

Determines whether a delay time should fire once or continue delaying and firing until descheduled. This value has no effect if the time specification is absolute.

`setRegular(boolean)`
`isRegular()`

Determines whether a repeating delay time measures from the time the event fires (regular) or from the time the event listeners finish executing (not regular). For example, imagine you have a timer specification that repeats every 10ms, but calling the listeners via `notifyListeners` takes 5ms. If the specification is regular, the listeners will be called every 10ms. If the specification is not regular, the listeners will be called every 15ms (10ms after the listeners finish executing). Due to system load or the length of time the event listeners take to execute, it may not be possible for regular specifications to fire fast enough. In this case, the specification will fire as fast as possible.

`setTime(long)`
`getTime()`

Sets the time at which the specification fires. Absolute times are in milliseconds since midnight, January 1, 1970 UTC. Relative times are in milliseconds. A specification with an absolute time less than `java.lang.System.currentTimeMillis()` will fire as soon as it is registered.

`addTimerWentOffListener(TimerWentOffListener)`

Adds a listener to call when the specification fires.

`removeTimerWentOffListener(TimerWentOffListener)`

Removes a listener from the list of listeners to call when the specification fires. If the listener is not currently registered on this timer specification, this method does nothing.

`setAbsoluteTime(long)`

Sets the timer specification to fire at the specified absolute time.

`setDelayTime(long)`

Sets the timer specification to fire once after the specified delay.

`notifyListeners(Timer)`

Calls all the listeners registered on this specification. This method is primarily for the benefit of implementers of subclasses of `Timer` and will normally not be used by developers.

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.

An implementation of timers will be provided so that developers can use the `sunw.util` timing classes with downloaded applets on non-PersonalJava platforms.

Compatibility for JDK 1.1-based Systems

The following APIs are provided to be included with downloaded applets to allow them to run on BusinessJava platforms that lack the new features in PersonalJava:

sunw.awt

```
package sunw.awt;

public interface DoubleBuffering { ... }

public interface NoInputPreferred {}
public interface KeyboardInputPreferred {}
public interface ActionInputPreferred {}
```

```
public interface PositionalInputPreferred {}
```

sunw.lang

```
package sunw.lang;

public class UnsupportedOperationException extends RuntimeException {
    ...
}
```

sunw.util

```
package sunw.util;

public abstract class Timer { ... }

public class TimerSpec { ... }

public class TimerWentOffEvent { ... }

public interface TimerWentOffListener { ... }
```

Networking Protocols

The PersonalJava networking classes support a wide variety of protocols, listed below. Licensees are free to use the implementations we supply, to use their own, or to leave optional ones out. Those marked as "required" are ones that applets universally assume to be available. We recommend that implementors include those marked as "optional" if space and the specifics of the device or system allow it.

The **Version** column lists the lowest version we recommend (or require, as appropriate) supporting.

The **Supplied** column details what we will include in the initial implementation of PersonalJava, including a version number if appropriate.

Name	Version	Required	Optional	Supplied
http:	1.0	✔		1.1
SSL (Secure Sockets Layer)	3.0		✔	✔
gopher:	--		✔	✔
ftp:	--		✔	✔ (requires `java.net.ServerSocket`)
mailto: (SMTP)	--		✔	✔
file:	--		✔	✔

Image Formats

PersonalJava assumes the ability to handle the set of graphical image formats listed below.

Where a version number is listed, it represents the lowest version we recommend (or require, as appropriate) supporting.

Format	Version	Required	Optional
CompuServe GIF	89a	✔	
JPEG (JFIF)		✔	
XBM (XBitmap)		✔	

XBitmap format isn't often used in programs, but is commonly used to represent icons in web server directory listings. It's a simple textual format that requires a very small amount of code to implement.

Note: Although a PersonalJava implementation is required to support the image formats listed above, different implementations may deliver the image data using different variations of the `ImageProducer` and `ImageConsumer` interfaces. For example, many implementations will deliver a GIF image using an instance of `IndexColorModel` to describe the colors. However, others may deliver the data using other color models. In particular, television based platforms may deliver data in a YUV (luminance/chrominance) color model. Likewise, some implementations may deliver a single scanline at a time, whereas others may deliver many scanlines at the same time, may deliver an entire image at once, or may deliver it in pieces not based on scanlines at all. Developers should always write according to the whole `ImageProducer` and `ImageConsumer` interface and not take shortcuts based on empirical results on a single platform.

Items Still Under Investigation

- None at present. Defer applet requirement declaration API until a later release.

Sun Microsystems, Inc.

2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.