



C H A P T E R 4

The Accessibility API

- 4.1 Using the Accessibility API 73
- 4.2 Understanding the Accessibility API 75
- 4.3 Accessibility through pluggable look and feel 79
- 4.4 The end result 79

Accessibility is all about understanding your customer and meeting their needs. All user interface design is geared towards making an application easy to use. Accessibility takes that ease-of-use idea up the next logical step in making interfaces usable by all people, regardless of any disability which prevents them from using a standard, traditional means of interaction.

Accessibility is a tremendously important issue, but also one almost universally overlooked by applications developers. The presence of a standardized accessibility API will be a tremendous asset in getting Java adopted by industries such as banking, telecommunications, and consumer electronics. These industries in particular have requirements for accessibility that have been created by laws or an awareness of the substantial size of the potential market for accessible applications.

Accessibility opens a whole new market to a product. It is estimated that the market for accessible products is approximately 40 million people in the United States alone. Considering that the population of Canada is (as of 1998) just over 30 million people, a market of 40 million people is a very healthy one to target.

Beyond the market size issues, it is interesting that programs that are accessible also tend to be easier for fully able users to use. Working with accessibility requires thinking about ideas such as keyboard access and reducing the number of steps to perform an action. Both of these make your application easier for everyone.

The term “accessibility technology” covers a very wide range of devices. The best way to characterize them is by describing the issues they are designed to deal with. There are three broad categories of disability:

- *Vision* Users with sight limitations need support in several areas. Both low- and no-sight users need keyboard means of performing any operation that would normally be done with a mouse. The reason for this isn't that a keyboard is easier for the user, it is that keyboard-style input is easier for a speech-style input system to use. This area is the easiest to test for, since you need only to unplug your mouse and try to use the application. Similarly, blind or low-vision users also need to know where the focus is and when it changes, as well as what the default action is and how to trigger it. These users also need text descriptions for both graphics and video. For videos it is important if an action is being shown to also describe it. Online documentation is also advantageous to both of these types of users, since it is much more convenient using text-to-speech than having to have Braille or large-print versions of paper documentation (for both the developer and the user).

Low-vision users can also benefit from screen magnifiers and high-contrast color Themes.

- *Hearing* Restricted-hearing users' needs can be summarized in a simple concept: don't use only sound to indicate something important. Don't just have a ringing noise for an incoming call; have a visual cue as well, such as flashing the “answer” icon. Text descriptions of any audio are also useful. Video should provide the information needed by a closed-captioning style service.

- *Mobility* Mobility-impaired users have needs that are less in the area of the application developer and more in the area of the assistive-technology vendor. The primary responsibility of an applications vendor is to provide a simple, rational system for choosing actions. Actions should not be buried so deeply it takes a concerted effort for a user to get to them. The last sentence should have caught your attention, since it is good user interface design applicable to *all* users. Online documentation is also beneficial to this category of user.

Supporting all of these variations may seem like a daunting task, so you'll probably be happy to find out it isn't completely your responsibility to implement all of it. Because Swing components support the Accessibility API, you are responsible only for providing the *information* needed by these technologies. Implementing the actual technologies is the responsibility of accessibility technology companies. You still have some details to think about, but consider it as simply paying more attention to the usability of your application.

Java is a significant improvement over most previous technologies in that its accessibility requirements are being considered very early in its life. Legacy operating systems generally added accessibility support much later in their life cycles (with the exception of the Macintosh, which did address many of these issues early on). Java also benefits from a full object-oriented structure, in which components are already subdivided into easily understandable parts. Java does not have to support applications written in procedural languages, unlike the existing operating system APIs. Swing is also a significant improvement for Java, since it mostly eliminates the native peers of the AWT. The transition from JVM to native component meant that accessibility technologies previously lost access to a significant amount of information.

Java isn't perfect in this area, and it is still evolving, but the cost to a developer of supporting accessibility is much lower than ever.

4.1 *Using the Accessibility API*

Making a Swing application accessible requires about the same degree of effort as internationalization. The application developer is responsible for providing the information that will be presented to the user by appropriate accessibility technologies.

Since `JComponent` implements `Accessible` (p.85), all Swing components are at least partially accessible. At its simplest, all that needs to be done to support accessibility with a Swing component is to set the `accessibleName` or `accessibleDescription` properties, or both.

For components that display text titles such as a `JLabel` or the tabs in a `JTabbedPane`, the `accessibleName` property will be set by default to the component's text. Situations where just an `Icon` is displayed (for example a `JButton` with just its `Icon` set) will require explicitly setting the `accessibleName`. If you ever wondered what the second parameter in the constructor for `ImageIcon` (p.184) was, your answer is that it is the information provided to the accessibility engine to describe the `Icon` (see figure 4.1).

```
printButton = new JButton(iconOfAPrinter);
printButton.setToolTipText("Print the current document");
printButton.getAccessibleContext().setAccessibleName("Print");
printButton.setMnemonic('p');
```

Figure 4.1 Setting the basic Accessibility information for an Icon-only JButton

Conveniently, the developers of Swing chose to make the `ToolTip` text of a control (if any) the default `accessibleDescription`. So, if you are already supplying `ToolTips` for your controls, you are pretty much finished setting the description. If a `ToolTip` is not supplied, the `accessibleDescription` needs to be explicitly set using `AccessibleContext`'s (p.90) `setAccessibleDescription()` method. `JLabels` (p.264) are a special case in that they have a `setLabelFor()` method that tells the accessibility engine what component this `JLabel` is describing. The `setLabelFor()` method information is also used by accessibility focus management so it can put the focus in the correct component (since the `JLabel` may be above or to the side of its associated component).

Naming `JPanels` is actually quite helpful. You, as the applications developer, are using the `JPanel` to group together related controls. Naming the panel and giving it a description (based on why you are grouping the controls it contains) provides an excellent way to give a disabled user more information about the purpose of the components it contains. Containers with a `TitledBorder` will use the text from the title as their name.

Swing 1.1 supplies keyboard support for most controls. The keys bound to the different actions are implemented by the installed look and feel. This leaves adding accelerator keys and mnemonics to the developer. As a rule, *all* menu items should have a way to select them using the keyboard. This support does not need to be a full accelerator keystroke, but should, at the very least, be a series of menu items with mnemonic characters leading to the desired menu item. Also, toolbar or floating palette buttons should have menu items that correspond to them and which also pass the keyboard access test.

Focus management needs to be implemented as well. The rule of thumb in this case is to have the focus on the component with which the user first needs to interact. If the activity in a panel is entering a name, have the focus placed in the text area for the first part of the name when the panel appears. A blind user can't use the mouse to click in a text area to establish focus. The default means Swing has of identifying the next component after the current one is the order in which components were added to the container. So, make sure that the order in which `add()` is called is consistent with the order that the focus should traverse the components. If you want to explicitly override the default component the focus will change to, `JComponent` supplies the `setNextFocusableComponent()` method. `JComponent` also supplies the `isFocusTraversable()` method, which can disable the ability of a component to get the focus if it returns `false`. You should not override this to return `false` in a subclass unless the component does not

have an associated action or you have explicitly supplied a keyboard means of triggering the action. Otherwise a user without a mouse would not be able to use the component.

Notifying the user about status or errors (through a `JDialog` or some form of status bar) should be implemented using Swing components so that the accessibility engine can notice these events and report them to the user.

4.2 Understanding the Accessibility API

The Accessibility API provides several classes and interfaces for use by traditional applications. The main interface is the `Accessible` interface (p.85).

The `Accessible` interface is very simple, containing the `getAccessibleContext()` method. The `AccessibleContext` (p.90) is the core of the Accessibility API, providing methods (table 4.1) to get all of the other information required. As such, it contains a large number of methods. Don't let that overwhelm you, since most of the methods are implemented to simply return `null`, indicating that the component does not support that kind of accessibility information. For example, a `JScrollBar`'s associated `AccessibleContext` will return an `AccessibleValue` (p.104) object from its `getAccessibleValue()` method, but it will return `null` from its `getAccessibleText()` method. `JLabel` is just the opposite, in that it returns an `AccessibleText` (p.103) object from its `getAccessibleText()` method and `null` from its `getAccessibleValue()` method. Most components in Swing implement their `AccessibleContext` using an inner class.

Table 4.1 Methods in `AccessibleContext` supporting accessibility features

| Method | Description |
|--|---|
| <code>getAccessibleAction()</code> | Returns the <code>AccessibleAction</code> (p.86) associated with this component. |
| <code>getAccessibleChild(int index)</code> | Returns the <code>Accessible</code> child at <code>index</code> in the component. |
| <code>getAccessibleChildrenCount()</code> | Returns the number of <code>Accessible</code> children in the component. |
| <code>getAccessibleComponent()</code> | Returns the <code>AccessibleComponent</code> (p.87) associated with this <code>AccessibleContext</code> . |
| <code>getAccessibleDescription()</code> | Returns the <code>accessibleDescription</code> property of this object. |
| <code>getAccessibleIndexInParent()</code> | Returns the 0-based index of this component in its <code>Accessible</code> parent. |
| <code>getAccessibleName()</code> | Returns the <code>accessibleName</code> property of this component. |
| <code>getAccessibleParent()</code> | Returns the <code>Accessible</code> parent of this component. |
| <code>getAccessibleRole()</code> | Returns the role of this component. |
| <code>getAccessibleSelection()</code> | Returns the <code>AccessibleSelection</code> (p.98) associated with this component. |
| <code>getAccessibleStateSet()</code> | Returns the <code>AccessibleStateSet</code> (p.101) of this component. |
| <code>getAccessibleText()</code> | Returns the <code>AccessibleText</code> (p.103) associated with this component. |

Table 4.1 Methods in `AccessibleContext` supporting accessibility features (continued)

| Method | Description |
|---|--|
| <code>getAccessibleValue()</code> | Returns the <code>AccessibleValue</code> (p.104) associated with this component. |
| <code>setAccessibleDescription(String description)</code> | Sets the localized <code>accessibleDescription</code> of this component. |
| <code>setAccessibleName(String name)</code> | Sets the localized <code>accessibleName</code> of this component. |
| <code>setAccessibleParent(Accessible parent)</code> | Sets the <code>Accessible</code> parent of this component. |

`AccessibleContext` supports `PropertyChangeListeners` as well. It has a number of property events (table 4.2) that it can generate so that accessibility technologies can be informed when the component changes; conversely, the component can be updated when the accessibility technology causes a change.

Table 4.2 Property names defined by `AccessibleContext`

| Property | Description |
|--|--|
| <code>ACCESSIBLE_ACTIVE_DESCENDANT_PROPERTY</code> | Property name used when the focused child in a <code>JTable</code> , <code>JList</code> , or <code>JTree</code> changes. |
| <code>ACCESSIBLE_CARET_PROPERTY</code> | Property name used when the <code>accessibleText</code> caret has changed. |
| <code>ACCESSIBLE_CHILD_PROPERTY</code> | Property name used when accessible children are added to or removed from the object. |
| <code>ACCESSIBLE_DESCRIPTION_PROPERTY</code> | Property name used when the <code>accessibleDescription</code> property has changed. |
| <code>ACCESSIBLE_NAME_PROPERTY</code> | Property name used when the <code>accessibleName</code> property has changed. |
| <code>ACCESSIBLE_SELECTION_PROPERTY</code> | Property name used when the <code>accessibleSelection</code> has changed. |
| <code>ACCESSIBLE_STATE_PROPERTY</code> | Property name used when the <code>accessibleStateSet</code> property has changed. |
| <code>ACCESSIBLE_TEXT_PROPERTY</code> | Property name used when the <code>accessibleText</code> has changed. |
| <code>ACCESSIBLE_VALUE_PROPERTY</code> | Property name used when the <code>accessibleValue</code> property has changed. |
| <code>ACCESSIBLE_VISIBLE_DATA_PROPERTY</code> | Property name used when the visual appearance of the object has changed. |

The `accessibleName` property is the name of the object. It is usually implemented to return the text contained in the control, if it has any. Since the contained text is displayed, it should be localized. For example, a “yes” button would have “Yes” as its `accessibleName` in the `en_US` (English-US) locale and “Oui” in the `fr_FR` (French-France) locale.

For complex controls that contain a number of items, such as a `JTable` or `JList`, the `accessibleName` needs to be explicitly set using the `setAccessibleName()` method. A `JTable` displaying the contents of a company’s orders database could have a localized `accessibleName` such as “Orders” in the `en_US` locale.

When the `accessibleName` property changes, an `ACCESSIBLE_NAME_PROPERTY` `PropertyChangeEvent` is fired.

The second important property is `accessibleDescription`. This property is a localized description of an object's purpose. The `JTable` from the `accessibleName` example could have an `accessibleDescription` such as "The Wallace Company's Grommet Order Database." By default, Swing components will use their `ToolTip` text (if they have any) as their `accessibleDescription`.

Changing the `accessibleDescription` property causes an `ACCESSIBLE_DESCRIPTION_PROPERTY` `PropertyChangeEvent` to be fired.

An `AccessibleValue` (p.104) tends to be used for classes that represent a range of values, such as a `JScrollBar` or `JSlider`. There are a few exceptions to this rule, which can be understood once you see what the value is representing. An `AbstractButton` (p.118) subclass uses its `accessibleValue` property to represent its pressed state. A `JSplitPane` (p.346) uses it to represent the location of the divider. Other components that use the associated `AccessibleValue` (p.104) interface are `JInternalFrame` (p.254), `JInternalFrame.JDesktopIcon` (p.263), and `JProgressBar` (p.315). The associated `PropertyChangeEvent` to this property is `ACCESSIBLE_VALUE_CHANGE`.

`AccessibleSelection` (p.98) is a collection of the selected `Accessible` children in the component. `AccessibleSelection` maintains an internal vector of selected items. The last item in this vector (at index `n-1`, since the vector is 0 based) is the last selected item. `AccessibleSelection` is extremely important to the user. It tells the user what is currently selected and it is supported by any component that allows its children to be selected, such as `JLists`, `JMenus`, or `JTrees`. A `JDesktopPane` should be subclassed to implement this interface when it is managing several child windows (it does not support this interface by default). `AccessibleSelection` does not handle text selections. Use the `AccessibleText` interface for that kind of selection.

`AccessibleComponent` (p.87) is the `Accessible` component associated with this `AccessibleContext`. `AccessibleComponent` defines a standard interface for the accessibility engine to directly interact with the component. The methods it implements mirror those found in `JComponent`.

`AccessibleText` (p.103) provides support for complex text manipulation. This interface also provides methods for controlling the text `Caret` when the text is editable. This interface is not for use by simple, noneditable components that contain text. For example, `JTextField`'s `AccessibleContext` returns an instance of an object supporting this interface, but `JButton` and `JLabel`'s `AccessibleContexts` return `null` from `getAccessibleText()`. The components that can be manipulated using this interface are the components that have a `Document` as their model. As long as any `Document` you create uses the `StyleConstants` (p.865) attributes, the component using the `Document` will be able to be manipulated through this interface. Note that text selections are manipulated through this interface, not the `AccessibleSelection` interface. The property changes fired by changes made using this interface are the `ACCESSIBLE_TEXT_PROPERTY` (content changes), `ACCESSIBLE_SELECTION_PROPERTY` (selection changes),

and `ACCESSIBLE_CARET_PROPERTY` (the `Caret` moved). Even though it is a change in the appearance of the component, changing the text contents of a `Document` does not fire the `ACCESSIBLE_VISIBLE_DATA_PROPERTY` event.

Since hypertext has become so common, a subinterface of `AccessibleText` is supplied by the Accessibility API. The `AccessibleHypertext` (p.94) interface has methods to simplify working with hypertext documents. Objects implementing this interface are still retrieved using the `getAccessibleText()` method. You can test the returned object using `instanceof` to see if it supports `AccessibleHypertext` once you have a reference to it.

The `accessibleParent` and `accessibleChildren` properties are handled transparently for you by `Swing`. As `Swing` components are added to and removed from each other, the appropriate `accessibleParent/accessibleChild` relationships are maintained.

The `accessibleRole` property of an object is a generic way of expressing what an accessible object is. The constants for the different roles are all defined in the `AccessibleRole` (p.95) class. Note that a single object can only have one role. A `JComboBox` has a role of `AccessibleRole.COMBO_BOX`, not `AccessibleRole.TEXT` and `AccessibleRole.LIST`. If you are subclassing a component for a more specific purpose, do not define your own `AccessibleRole`. Use the `accessibleDescription` to distinguish the new component from its parent class instead. The reason for this is that the accessibility technology doesn't care that the purpose of the component is different as long as it still behaves in roughly the same way.

The `accessibleState` property of an object expresses its current state. For example, a `JCheckBox` can be checked or not checked. A single object may have many `AccessibleStates` (p.99) and they are all combined into an instance of the `AccessibleStateSet` (p.101) class. The previously mentioned `JCheckBox` may also have the `AccessibleState.ENABLED` and `AccessibleState.FOCUSED` states in its `AccessibleStateSet` along with the `AccessibleState.CHECKED` state.

Components that trigger an `Action` will have an `AccessibleContext` that returns an `AccessibleAction` (p.86) object. An `AccessibleAction` contains a list mapping between actions and their descriptions. The list will be presented to the user, and the user can then choose to trigger one of the actions by its index. This makes it possible to use `JToolBar` and `JMenu` without a mouse. Web pages can present their image maps in the form of an `AccessibleAction` as well, where each image map hotspot is represented as an entry in the list. The description would indicate where the hotspot would take the user and the associated action would be triggering the hyperlink. In fact this is a common-enough usage of `AccessibleAction` that there is a subinterface of `AccessibleAction` named `AccessibleHyperlink` (p.93). The `getAccessibleAction()` interface is used to retrieve objects that implement `AccessibleHyperlink`. Once you have a reference to an object returned by `getAccessibleAction()` you can test it (using `instanceof`) to see if it implements the extended `AccessibleHyperlink` interface.

4.3 Accessibility through pluggable look and feel

A second way accessibility can be implemented is through writing a custom look and feel. This method is appropriate for situations where the user interface needs radical changes to support the accessibility technology, such as a pure speech interface, or a puffer/headtracker interface.

A full, new look and feel could be written to support a new technology, but a much better solution has been provided. As discussed in the chapter on look and feel, the Multi look and feel is designed primarily to support accessibility technologies. If you are trying to provide a new input mechanism for the user, all you need to implement using the Multi look and feel is the input part of the look and feel. You can use an existing look and feel (such as Metal) for the visual output. A look and feel designed for the Multi look and feel is known as an auxiliary look and feel. Auxiliary look and feels can also be transparently combined, so an input look and feel can be combined with an output look and feel (such as a screen reader).

For more information about how the Multi look and feel works, see chapter 3.

4.4 The end result

Accessibility, like internationalization, may seem at times to be more trouble than it is worth. With accessibility, you are enabling people to use your product who may not have *any* other choices. That's worth a lot.

Beyond the warm fuzzy reasons, there are also sound business reasons. There are very large lobby groups committed to accessibility, and very large industries that have already seen the positive business aspects to supporting these technologies. That is even beyond having a market which wants applications and until now hasn't had many choices.

Swing makes it relatively easy to support accessibility in your applications. Since you are already doing most of the work of supporting accessibility through adding text and ToolTips to your application (without even knowing it), doing the rest of the work is well worth it.

