Accessibility Design and Coding Guidelines for Java Swing and SWT GUI Development

by

Barry Feigenbaum, Ph. D.
IBM Worldwide Accessibility Center

Updated by

Sueann Nichols
IBM Human Ability and Accessibility Center

Table of Contents

Figure List

1 Introduction

1.1 What is this document about

This document is a supplement to the existing *Software accessibility* checklist and the *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java* document, all from the IBM Worldwide Accessibility Center (see Resources). It does not replace these documents. They are considered to be prerequisites to using this document.

This document is intended to assist developers in meeting the accessibility requirements listed in the Software accessibility checklist. It is further intended to give guidance to architects and developers conducting design, code reviews or both for graphical user interface (GUI) implementations using the Java[1] language. In particular this document provides guidance for developers using the Java *Swing* and Eclipse *Standard Widget Toolkit* (SWT) GUI libraries.

It is hoped that by using the guidelines in this document any design or code review will discover as many areas as possible where accessibility support is missing or weakly implemented—and discover them prior to test or deployment so they can be addressed economically. Included are programming examples on how to add select accessibility support for the particular design and code guidelines discussed.

This document does not provide rules and procedures for conducting code or design reviews—those should be developed by your team. You should have design artifacts, code artifacts or both available for inspectors when conducting any reviews. If you have reached the point where you can run your GUI you may also want to bring reports from tools like *Scout* and *Inspect* (see Verification tools) to help your inspectors evaluate your GUI.

This document is not a Swing or SWT tutorial. It is assumed the reader is familiar with Swing development, SWT GUI development or both.

This document does not specify any accessibility compliance criteria; that task is left up to other documents and standards. The word "ensure" is used below to indicate that you should consider implementing the guideline, not that the guideline is necessary (or sufficient) to meet some particular accessibility criteria.

All development processes that support Java GUI development should include a step or checkpoint statement similar to the following:

[1] Although Java centric, this document can be (easily) adapted for guidance for other languages running in the Java Runtime Environment.

"The IBM Corporate Instruction 162 (and some government agencies we market to) requires all user interfaces for IBM products and tools to be enabled for accessibility to support persons with disabilities. Thus any new or changed GUI code must meet established standards for accessibility enablement."

This document is about specific checks and enablement techniques that can be made to help reviewers and developers create that code meets this checkpoint.

1.1.1 Software Accessibility checklist

If you look at the *Software Accessibility* checklist you will see the major areas that should be addressed. I will discuss them at a high level now and detail checks related to them in the body. From the checklist:

Application developers should consider the special needs of users with disabilities when they develop software programs in Java. Meeting the needs of users with disabilities will benefit all users. In these guidelines, we discuss the following topics:

- *Essential accessibility programming practices that make applications usable by people with disabilities.*
- *The Java Foundation Classes (JFC), also known as Swing, and their implementation of accessibility methods in the user-interface components. Application developers should take advantage of this easy way to incorporate accessibility into their designs.*
- *The Java Accessibility application programming interface (API). If developers choose to create components themselves, they should implement this API.*

The following is the current[2] Software Accessibility checklist. For more information see Resources.

1	***Keyboard Access***	***Related Guidelines***
1.1	Provide keyboard equivalents for all actions.	Ensure you create accelerators for key Controls. Ensure a valid tab order is established.

[2] June 16, 2004 version.

1.2	Do not interfere with keyboard accessibility features built into the operating system.	Ensure you can adjust attributes of your GUI.

2	***Object information.***	***Related Guidelines***
2.1	Provide a visual focus indicator that moves among interactive objects as the input focus changes. This focus indicator must be programmatically exposed to assistive technology.	Ensure some control gets focus
2.2	Provide semantic information about user interface objects. When an image represents a program element, the information conveyed by the image must also be available in text.	** ALL ***
2.3	Associate labels with controls, objects, icons and images. If an image is used to identify programmatic elements, the meaning of the image must be consistent throughout the application.	Establish inter-Control relationships
2.4	When electronic forms are used, the form shall allow people using assistive technology to access the information, field elements and functionality required for completion and submission of the form, including all directions and cues.	** NONE **

3	***Sound and multimedia***	***Related Guidelines***
3.1	Provide an option to display a visual cue for all audio alerts.	Use multi-modal indicators.
3.2	Provide accessible alternatives to significant audio and video.	Use multi-modal indicators.
3.3	Provide an option to adjust the volume.	Ensure you can adjust attributes of your GUI.

4	***Display***	***Related Guidelines***
4.1	Provide text through standard system function calls or through an API (application programming interface) which supports interaction with assistive technology.	
4.2	Use color as an enhancement, not as the only way to convey information or indicate an action.	Use text attributes only for embellishment, not information. Ensure you provide

		accessible content.
4.3	Support system settings for high contrast for all user-interface controls and client-area content.	Ensure you can adjust attributes of your GUI.
4.4	When color customization is supported, provide a variety of color selections capable of producing a range of contrast levels.	Ensure you can adjust attributes of your GUI.
4.5	Support system settings for size, font and color for all user interface controls.	Ensure you can adjust attributes of your GUI
4.6	Provide an option to display animation in a non-animated presentation mode.	Use multi-modal indicators

5	***Timing***	***Related Guidelines***
5.1	Provide an option to adjust timed responses or allow the instruction to persist.	Avoid automatic advancement.
5.2	Avoid the use of blinking text, objects or other elements.	Ensure you can adjust attributes of your GUI

6	***Verify accessibility***	***Related Guidelines***
6.1	Test for accessibility using available tools.	** NONE **

Note that this document does not provide a one-for-one guideline to check item mapping. This document concentrates on the most critical aspects of accessibility enablement and many checklist items are grouped into a single guideline.

1.2 Terms used in this document

In this document we will be discussing both Swing- and SWT-based GUIs. These GUI toolkits use different terminology. To avoid confusion we will adopt the SWT terminology in this document. The following terms are used:

- **Widget**—a GUI part. In Abstract Windows Toolkit (AWT)/Swing this is called a *Component*.

- **Control**—a Widget with accessibility functionality. In SWT a Control has a host operating system Control associated with it. Equivalent to a *JComponent* in Swing.

- **Composite**—a Control that can contain other Controls. Equivalent to a *JPanel* in Swing. Composites are often also called *Containers*.

- **Canvas**—a Composite that can be drawn on. Equivalent to a JPanel in Swing.

1.3 Comparing SWT and Swing

This document discusses Swing and Eclipse/SWT development and the Swing and SWT accessibility APIs. This document will not discuss Java AWT-based GUIs (except as they provide a foundation for Swing GUIs) because AWT has incomplete support for accessible GUIs and thus should not be used to construct accessible GUIs.

Swing and SWT have certain differences in philosophy that can have an impact on accessibility and how programmers implement accessibility. Both Swing and SWT provide rich accessibility APIs. If you study them you can see a large degree of similarity in function with a significant difference in style. The differences are summarized below:

- Swing supports *in-process* assistive technologies.

  This means the assistive technology is running in the same JVM as the application it is processing. It further means all of the APIs (standard and assistive) provided by a Control are directly available to the assistive technology. Event callbacks and method calls execute in less elapsed time. Each `JComponent` subclass supports a parallel `Accessible-Context` hierarchy that provides access to accessible descriptions, states and behaviors of each `JComponent`.

- SWT supports *out-of-process* assistive technologies.

  This means the assistive technology must make fairly expensive cross-process calls to access information. As a result, fewer states and events are available than with Swing. Controls support an `Accessible` object that provides a mechanism, through registered listeners, to allow the

Control to override standard accessible states values reported to an assistive technology.

SWT also differs from Swing in how Controls are created. In SWT all Controls need a parent Composite at creation time. This forces a top-down construction of GUIs. In Swing, children Widgets are added to the parent. This means they can be created before or after the parent is created and added at any time. The Swing approach is more flexible.

In SWT most Control decorations—such as scrollbars and borders—are controlled by flags, called *style bits*. These style bits are provided when the Control is created and cannot be changed later. In Swing these decorations are independent of the Control and are added as Composite Widgets. The Swing approach is more flexible but often takes more code.

Both Swing and SWT provide significant accessibility by default. In Swing this support is coded directly into the Swing run-time. In general, support is consistent across all Java platforms. In SWT the host operating system accessibility features are used. Support can differ significantly across host platforms.

To the extent that you use standard Swing or SWT Controls and do not specifically set values, such as color and fonts sizes, the more the GUI will be enabled by default. Still, a high level of accessibility enablement cannot be fully achieved by default. You need to provide some accessibility support in your own code.

You should also take care when overriding mouse and keyboard listeners. When you do this you may disable the native support for accessibility (for example, "sticky-keys") that is applied by the run time. This is especially true for Alt- and Ctrl-shifted keys. Extra testing with assistive technologies is needed when these listeners are used. The *Software Accessibility Guidelines* provide more information on particular keys to use.

When you must go beyond the support provided by Swing and SWT (such as when you create custom Controls), you need to provide explicit accessibility enablement.

1.3.1 Overview of the Java Accessibility API.

The *Java Accessibility API* (JAA), in package `javax.accessibility`, is implemented in all [3] Java GUI Controls (although it is not fully implemented on all

[3] List of know implementers: Box, Box.Filler, Button, Canvas, CellRendererPane, Checkbox, CheckboxMenuItem, Choice, HTMLEditorKit, ImageIcon, JApplet, JButton, JCheckBox, JCheckBoxMenuItem, JColorChooser, JComboBox, JDesktopPane, JDialog, JFileChooser, JFrame, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JList.-

AWT Controls). Each `JComponent` subclass implements the `Accessible` interface which has one method - `getAccessibleContext()`. Through the returned `AccessibleContext` many queries can be performed and actions taken on the corresponding `JComponent`.

The `AccessibleContext` API is summarized [4] below:

Method Summary

void	**addPropertyChangeListener**(PropertyChangeListener listener) Adds a PropertyChangeListener to the listener list.
void	**firePropertyChange**(String propertyName, Object oldValue, Object newValue) Support for reporting bound property changes.
AccessibleAction	**getAccessibleAction**() Gets the AccessibleAction associated with this object that supports one o more actions.
abstract Accessible	**getAccessibleChild**(int i) Returns the specified Accessible child of the object.
abstract int	**getAccessibleChildrenCount**() Returns the number of accessible children of the object.
AccessibleComponent	**getAccessibleComponent**() Gets the AccessibleComponent associated with this object that has a graphical representation.
String	**getAccessibleDescription**() Gets the accessibleDescription property of this object.
AccessibleEditableText	**getAccessibleEditableText**() Gets the AccessibleEditableText associated with this object presenting editable text on the display.
AccessibleIcon[]	**getAccessibleIcon**() Gets the AccessibleIcons associated with an object that has one or more associated icons
abstract int	**getAccessibleIndexInParent**() Gets the 0-based index of this object in its accessible parent.
String	**getAccessibleName**() Gets the accessibleName property of this object.

AccessibleJList.AccessibleJListChild, JMenu, JMenuBar, JMenuItem, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRadioButton, JRadioButtonMenuItem, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSplitPane, JTabbedPane, JTable, JTable.Accessible-JTable.AccessibleJTableCell, JTableHeader, JTableHeader.AccessibleJTableHeader.Accessible-JTableHeaderEntry, JTextComponent, JToggleButton, JToolBar, JToolTip, JTree, JTree.-AccessibleJTree.AccessibleJTreeNode, JViewport, JWindow, Label, List, List.Accessible-AWTList.AccessibleAWTListChild, Menu, MenuBar, MenuItem, Panel, Scrollbar, ScrollPane, TextComponent, Window

[4] see the J2SE JavaDoc via a browser for a more detailed and accessible form of this summary.

Accessible	**getAccessibleParent**() Gets the Accessible parent of this object.
AccessibleRelationSet	**getAccessibleRelationSet**() Gets the AccessibleRelationSet associated with an object.
abstract AccessibleRole	**getAccessibleRole**() Gets the role of this object.
AccessibleSelection	**getAccessibleSelection**() Gets the AccessibleSelection associated with this object which allows its Accessible children to be selected.
abstract AccessibleStateSet	**getAccessibleStateSet**() Gets the state set of this object.
AccessibleTable	**getAccessibleTable**() Gets the AccessibleTable associated with an object.
AccessibleText	**getAccessibleText**() Gets the AccessibleText associated with this object presenting text on th display.
AccessibleValue	**getAccessibleValue**() Gets the AccessibleValue associated with this object that supports a Numerical value.
abstract Locale	**getLocale**() Gets the locale of the component.
void	**removePropertyChangeListener**(PropertyChangeListener listen Removes a PropertyChangeListener from the listener list.
void	**setAccessibleDescription**(String s) Sets the accessible description of this object.
void	**setAccessibleName**(String s) Sets the localized accessible name of this object.
void	**setAccessibleParent**(Accessible a) Sets the Accessible parent of this object.

1.3.2 Overview of the SWT Accessibility API

The SWT *Accessibility API* (SAA), in package `org.eclipse.swt.accessibility`, is implemented for all SWT GUI Controls (but not Widgets). Each Control provides an `Accessible` proxy (accessed with the `getAccessible()` method) to the host platform's accessibility APIs. Through the `Accessible` object many actions can be taken on the corresponding `Control`.

The `Accessible` API is summarized [5] below:

Method Summary

[5] See the J2SE JavaDoc via a browser for a more detailed and accessible form of this summary.

void	**addAccessibleControlListener**(AccessibleControlListener listener) Adds the listener to the collection of listeners who will be notified when an accessible client asks for custom control specific information.
void	**addAccessibleListener**(AccessibleListener listener) Adds the listener to the collection of listeners who will be notified when an accessible client asks for certain strings, such as name, description, help or keyboard shortcut.
void	**addAccessibleTextListener**(AccessibleTextListener listener) Adds the listener to the collection of listeners who will be notified when an accessible client asks for custom text control specific information.
Control	**getControl**() Returns the control for this Accessible object.
void	**removeAccessibleControlListener**(AccessibleControlListener listener) Removes the listener from the collection of listeners who will be notified when an accessible client asks for custom control specific information.
void	**removeAccessibleListener**(AccessibleListener listener) Removes the listener from the collection of listeners who will be notified when an accessible client asks for certain strings, such as name, description, help or keyboard shortcut.
void	**removeAccessibleTextListener**(AccessibleTextListener listener) Removes the listener from the collection of listeners who will be notified when an accessible client asks for custom text control specific information.
void	**selectionChanged**() Sends a message to accessible clients that the child selection within a custom container control has changed.
void	**setFocus**(int childID) Sends a message to accessible clients indicating that the focus has changed within a custom control.
void	**textCaretMoved**(int index) Sends a message to accessible clients that the text caret has moved within a custom control.
void	**textChanged**(int type, int startIndex, int length) Sends a message to accessible clients that the text within a custom control has changed.
void	**textSelectionChanged**() Sends a message to accessible clients that the text selection has changed within a custom control.

As you can see, this class supports a number of listeners. Through these listeners you can override or augment any default information provided by the platform's accessibility APIs, or create information for any custom Controls that you build.

The `AcccessibleListener` interface, used on all Controls, provides these methods:

Method Summary	
void	**getDescription**(AccessibleEvent e) Sent when an accessibility client requests a description of the control or a description of a child of the control.
void	**getHelp**(AccessibleEvent e) Sent when an accessibility client requests the help string of the control or the help string of a child of the control.
void	**getKeyboardShortcut**(AccessibleEvent e) Sent when an accessibility client requests the keyboard shortcut of the control or the keyboard shortcut of a child of the control.
void	**getName**(AccessibleEvent e) Sent when an accessibility client requests the name of the control or the name of a child of the control.

The `AcccessibleControlListener` interface, used on Composite, advanced or custom controls, provides these methods:

Method Summary	
void	**getChild**(AccessibleControlEvent e) Sent when an accessibility client requests the accessible object for a child of the control.
void	**getChildAtPoint**(AccessibleControlEvent e) Sent when an accessibility client requests the identifier of the control child at the specified display coordinates.
void	**getChildCount**(AccessibleControlEvent e) Sent when an accessibility client requests the number of children in the control.
void	**getChildren**(AccessibleControlEvent e) Sent when an accessibility client requests the children of the control.
void	**getDefaultAction**(AccessibleControlEvent e) Sent when an accessibility client requests the default action of the control or the default action of a child of the control.
void	**getFocus**(AccessibleControlEvent e) Sent when an accessibility client requests the identity of the child or control that has keyboard focus.
void	**getLocation**(AccessibleControlEvent e) Sent when an accessibility client requests the location of the control or the location of a child of the control.
void	**getRole**(AccessibleControlEvent e) Sent when an accessibility client requests the role of the control or the role of a child of the control.
void	**getSelection**(AccessibleControlEvent e) Sent when an accessibility client requests the identity of the child or control that is currently selected.
void	**getState**(AccessibleControlEvent e)

	Sent when an accessibility client requests the state of the control or the state of a child of the control.
void	**getValue**(AccessibleControlEvent e) Sent when an accessibility client requests the value of the control or the value of a child of the control.

The AcccessibleTextListener interface, used on select text Controls, provides these methods:

Method Summary

void	**getCaretOffset**(AccessibleTextEvent e) Sent when an accessibility client requests the current character offset of the text caret.
void	**getSelectionRange**(AccessibleTextEvent e) Sent when an accessibility client requests the range of the current text selection.

Note: some of these APIs are available only on Eclipse 3.0 or later.

1.4 Verification tools

Today there are not any [6] commercial tools that validate rich-client GUIs (i.e., native, not HTML, based GUIs) efficiently. The best approach available today is to use a screen reader product (such as *Job Access With Speech* by Freedom Scientific) and navigate your GUI using this tool, listening for the content it presents and comparing that content with the visual GUI. You should also use the screen reader with the display turned off to get a better understanding of how users who are visually impaired experience your GUI. You may also want to try to navigate your GUI with only a keyboard (without using the mouse). For more information on testing with JAWS see Resources.

The IBM Worldwide Accessibility Center provides a list of tools that can be used to assist with accessibility testing. See Resources for more information. In particular the IBM *Scout* tool (see http://w3-03.ibm.com/able/dl/#Scout) and some tools from Sun, such as *Monkey* and *Ferret*, can be used to inspect Swing-based GUIs. The Microsoft® *Inspect* tool (see http://w3-03.ibm.com/able/devtest/AC_AT_and_Tools.html#actr_insp) can be used to inspect SWT-based GUIs. These tools are highly interactive and, like screen readers, can be quite time consuming and tedious to use.

[6] To the author's knowledge.

For some operating systems—including Microsoft Windows®, which offers the *Microsoft Active Accessibility* (MSAA) feature—the platform accessibility APIs used by assistive technologies do not make available all the information that an assistive technology may need to present to the user. Many assistive technologies use additional (often back door or hack) techniques to access more information. This is particularly true for complex Controls like tables, browsers views, spreadsheets and word processing editors. Note that SWT on Windows is completely dependent on, and limited by, MSAA for its default accessibility.

All of the inspection tools mentioned above are limited because they use the architected platform accessibility APIs to access the information they display. For example, *Inspect* uses only the MSAA feature. These tools can lead you to believe your GUI is less accessible than it really is when accessed through a particular assistive technology. For this reason, IBM recommends using one or more assistive technologies as part of the process to verify your enablement.

1.5 Some additional thoughts

Enabling a GUI for ease of use for persons with disabilities often makes the GUI much easier for everyone to use. This can make your product more competitive against others in the industry. Creating accessible user interfaces is just good design and good business.

Enabling a GUI for accessibility is much like enabling an application for internationalization. You must take steps during design, coding and testing to ensure you have a good result. It is also far more expensive to add accessibility as an afterthought than to build it in from the beginning.

This document is a "living" document. It will be updated and revised frequently. Please make sure you are using the latest version available at the internal accessibility site (see Resources). New guidelines and examples will be added as they are identified; we welcome your suggestions.

2 Coding Guidelines

2.1 Control Creation Guidelines

Apply the checks in this section for each Control you create. Your code should perform these actions as soon as possible after creating the Control. We suggest coding these tasks in the code immediately after the Control is created. We further recommend you create a library of functions that you use to create your Controls. These library functions can then set many of these values (semi-) automatically.
The *Coding for Accessibility* article (see resources) gives more examples on how you can do this.

2.1.1 Ensure a valid tab order is established

For many people with disabilities, pointing devices like a mouse are not usable. For an efficient and usable interface, all input Controls and Controls that display critical information must be accessible using the keyboard. Keyboard users move between Controls primarily by use of the Tab or Reverse Tab (often Shift-Tab) keys, so all Controls need to be reachable via the Tab key. The exception is Controls that are part of an integrated group. The group must be accessible by Tabs but the Controls within the group can be reached by other keys, such as the cursor up, down, left and right keys (arrow keys).

The sequence of Controls selected by using Tab key is called the "tab order". All Controls that provide important user information or accept user input in a GUI must be on the tab order.

The Controls in the tab order must be selected in a way that makes sense to the user. Some example tab orders are:

- Left-to-right and top-to-bottom
- Row order in a table or list
- Column order in a table or column of items
- By group and then one of the above within the group

In SWT the tab order is primarily set by the order in which you add Controls to a Composite, so you should create the Controls in the desired order. You can override this by using the `Composite.setTabList` method.

In Swing, the default tab order is also the order you add Controls to a Composite, but the tab order can be overridden by creating a *FocusTraversalPolicy*.

Swing Example:

```
// create the top panel
JPanel mainPanel = new JPanel(new BorderLayout());

// create the data panel
JPanel dataPanel = new JPanel(new GridLayout(0, 2));
dataPanel.add(new JLabel("Name"));
dataPanel.add(new JTextField(40));
dataPanel.add(new JLabel("Address"));
dataPanel.add(new JTextField(40));
mainPanel.add(dataPanel, BorderLayout.CENTER);

// create the button bar
JPanel buttonPanel = new JPanel(new FlowLayout());
buttonPanel.add(new JButton("Next"));
buttonPanel.add(new JButton("Previous"));
buttonPanel.add(new JButton("Ok"));
buttonPanel.add(new JButton("Cancel"));
mainPanel.add(buttonPanel, BorderLayout.SOUTH);
```

Which generates a GUI like:

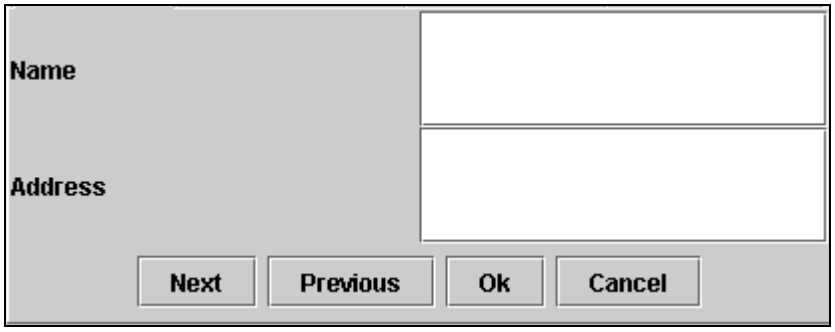

Figure 1 Swing GUI showing JTextFields and JButtons as defined above

Each Control receives focus in a left-to-right, top-to-bottom tab order, and is represented internally as:

	JPanel	panel	enabled,focus...					
	JPanel	panel	enabled,focus...					
	JLabel	label	enabled,focus...					'Name'
1	JTextField:NameField	text	enabled,focus...		page...			
	JLabel	label	enabled,focus...					'Address'
2	JTextField:addressField	text	enabled,focus...		page...			
	IPanel	panel	enabled,focus...					
3	JButton	push button	enabled,focus...	0	click			'Next'
	JButton 4	push button	enabled,focus...	0	click			'Previous'
5	JButton	push button	enabled,focus...	0	click			'Ok'
	JButton 6	push button	enabled,focus...	0	click			'Cancel'

Figure 2 Tree showing component containment and tab order

Note that each `JTextField` has a name. This is an example of setting a name (see Give each Control a unique identifier) using the code:
nameField.setName("NameField");
addressField.setName("addressField");

SWT Example

```
// create the top panel
Composite main = new Composite(parent, SWT.NONE);
GridLayout mgl = new GridLayout();
mgl.numColumns = 1;
main.setLayout(mgl);

// create the data panel
Composite data = new Composite(main, SWT.NONE);
GridData dgd = new GridData();
dgd.grabExcessHorizontalSpace = true;
dgd.grabExcessVerticalSpace = true;
data.setLayoutData(dgd);

GridLayout dgl = new GridLayout();
dgl.numColumns = 2;
data.setLayout(dgl);

Label l1 = new Label(data, SWT.NONE);
l1.setText("Name");
Text t1 = new Text(data, SWT.SINGLE);
GridData t1gd = new GridData();
t1gd.grabExcessHorizontalSpace = true;
t1.setLayoutData(t1gd);

Label l2 = new Label(data, SWT.NONE);
l2.setText("Address");
Text t2 = new Text(data, SWT.SINGLE);
GridData t2gd = new GridData();
t2gd.grabExcessHorizontalSpace = true;
```

```
t2.setLayoutData(t2gd);

// create the button bar
Composite buttons = new Composite(main, SWT.NONE);
RowLayout rl = new RowLayout();
buttons.setLayout(rl);
Button b = null;
b = new Button(buttons, SWT.PUSH);
b.setText("Next");
b = new Button(buttons, SWT.PUSH);
b.setText("Previous");
b = new Button(buttons, SWT.PUSH);
b.setText("Ok");
b = new Button(buttons, SWT.PUSH);
b.setText("Cancel");
```

The above sequence causes the tab order to be the name field, address field and then the next, previous, ok and cancel buttons (label Controls do not receive focus so they are not included in the tab order).

Some complex Controls that contain other Widgets often use the arrow keys (i.e., left, right, up, down), and not the Tab key, to move between Widgets within the Control. Typically this is automatic, but if you must implement this motion in your code, you need to ensure that the movement between Widgets is predictable and logical.

Occasionally another key, such as the use of the Alt key to switch to or from a menu bar, can be used to move between major groups of Controls.

2.1.2 Ensure some control gets focus

For people who are visually impaired, it is critical that they can determine at all times where they are in a GUI. To do this, there must always be some Control that has focus. This is especially true when a major context switch, such as selecting a different application window, occurs.

On all Composites, you need to ensure that some Control in the Composite gets focus whenever the Composite receives focus. For Swing and SWT this is typically automatic but there may be problems when the Composite is first presented or is re-shown after being unhidden.

The basic requirement is to add listeners to each top-level Composite so that when it becomes active (e.g., when it or one of its, possibly indirect, children gets focus) then an appropriate child Widget must be given focus. In general it is not sufficient for a Composite itself to receive focus, one of its child input Controls must receive focus.

Swing Example:

```
// very primitive example with just one button
final JFrame frame = new JFrame("My Frame");
final JButton button = new JButton("Press Me!");
frame.getContentPane().add(button);

// add a Window Listener to the JFrame that requests the
// button to have focus when the window is activated;
frame.addWindowListener(new WindowAdapter() {
   public void windowActivate() {
      button.requestFocus();
   }
}
:
frame.setVisible(true); // show the GUI
```

You need to add the window listener if focus has never been set before in your window. Swing remembers the last component with focus set.

SWT Example:

```
// very primitive example with just one button
final Display display = new Display();
final Shell shell = new Shell(display, SWT.TRIM);
shell.setText("My Shell");
final Button button = new Button(shell, SWT.PUSH);
button.setText("Press me!");

// add a Shell Listener to the Shell that requests the
// button to have focus when the window is activated.
shell.addShellListener(new ShellAdapter() {
    public void shellActivated(ShellEvent se) {
        button.setFocus();
    }
});
:
shell.open();    // show the GUI

// standard SWT event dispatch loop
while (!shell.isDisposed()) {
    if (!display.readAndDispatch()){
        display.sleep();
    }
}
```

```
display.dispose();
```

You need to add the shell listener if focus has never been set before in your window. The host GUI system remembers the last component with focus set.

If the Control you want to receive focus is a button, you can use code like the following instead:

Swing Example:

```
JFrame frame = new JFrame();
JButton button = new JButton("Press Me");
:
// give the button the default focus
frame.getRootPane().setDefaultButton(button);
:
```

Examples above based on those in the *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java*.

2.1.3 Give each Control a unique identifier

Being able to identify GUI Controls is important. It can help generation tools, debuggers and any programmer that looks at the Widget tree identify different Controls. It can help assistive technologies in presenting a good description of the Control. It can also be useful to automatically apply updates to a GUI, say from values in a properties file by using the identifier as part of a properties key.

In Swing, each Control has a name set by use of the `setName` method. The default names, if any, are fairly nondescript. We recommend that you explicitly set a unique name based on the Control's role in the GUI. For example, a Text Field used to enter a user name could be called the "usernameField". Names should be unique to the entire GUI, but if that is not possible then they must be unique to the parent Composite and the Composite needs to have a unique name in its Composite (and so on).

Supports for Control names is not defined in SWT. You can use the Widget `setData(key, value)` method to give each Widget (and thus Control) a name.

Swing Example:

```
// create field with name
JTextField field = new JTextField(20);
```

```
parent.add(field);
field.setName("userNameField");
```

See the examples for Establish inter-Control relationships.

SWT Example:

```
// create field with name
Text field = new Text(parent, SWT.SINGLE);
field.setData("name", "userNameField");
```

2.1.4 Ensure all text is seen

Users who are visually impaired need to be able to access the same information presented to sighted users. All GUI content must be identifiable to the assistive technology they are using.

A common situation is to have a paragraph of instructions at the start of an entry dialog. Many assistive technologies skip standalone "static" text (e.g., labels) by default. This means if you use labels that are not associated with other Controls that can receive focus, they will not be presented to the user by default. If the information is critical to using the GUI, you need to make sure the text is presented by the assistive technology.

One way to do this is to use a read-only text field for the text. Since the field will receive focus, its text will be read. However, this option can have undesirable visual effects. Another approach is to make sure the label is associated with some field that does receive focus.

Some assistive technologies may be able to be configured to look for standalone static text in some circumstances and present it (e.g.,a “read all” mode). If you use a lot of descriptive static text you may need to provide instructions in your product's help text on how to get popular assistive technologies to process static text. If you know the assistive technology your users will use has this feature, then you do not need to ensure that all static text is read. Still, if the text is critical to completing the GUI (such as introductory instructions), it should be included in the normal content an assistive technology presents.

Swing Example:

```
// create text area to provide static text
JTextArea description = new JTextArea(10, 40);
description.setText("Some long explanation …");
description.setEditable(false);
```

```
parent.add(description);
```

This example generates a GUI like:

Figure 3 Swing GU showing static text in a non-editable JTextField

The text is displayed as if it was static text, but since the JTextArea Control receives focus, the text is read by an assistive technology.

SWT Example:

```
// create text area to provide static text
int style = SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL |
            SWT.WRAP | SWT.READ_ONLY;
Text description = new Text(parent,  style);
description.setText("Some long explanation …");
description.setEnabled(false);
```

Many Controls, such as text fields, do not have a good default description. When read by an assistive technology, they are often presented only with a phrase such as “text editable” followed by the text in the field. This description provides no information about what type of information is desired in the field. Therefore it is very highly recommended that any such Control be labeled (preceded by a label Control) that describes the type of information needed.

Swing Example:

```
// create text with associated label
JLabel laddress = new JLabel("Address: ");
parent.add(laddress);
JTextArea address = new JTextArea(10, 40);
parent.add(address);
laddress.setLabelFor(address);
```

SWT Example:

```
// create text with associated label
Label laddress = new Label(parent, SWT.LEFT);
laddress.setText("Address: ");
Text address = new Text(parent,  SWT.SINGLE);
```

2.1.5 Use text attributes only for embellishment, not information

Most assistive technologies either do not report text attributes or require the user to take extra steps to get this information. Using these attributes as the only indication of important Control state or dynamic information can result in changes to this state or information not being noticed by the user. Therefore do not depend on these modifications alone to indicate critical information; other indicators are also required.

Often text appearance is enhanced by setting font modifiers (such as *Italics* or **bold**) or different colors. Some examples are placing special marker text, such as an asterisk (*) near the special text, or rewording the content such that the wording alone indicates the emphasis. Any such indicators should be described in the GUI itself, such as by labels.

Swing Example:

```
// items in list and associated modified flags
String[] items = {"Item 1", "Item 2", …,  "Item N"};
boolean[] mod =  {false,    true,     …,  true};

// create list with modified indicators
JLabel llist = new JLabel(
     "Modified items are suffixed with \"*\"");
parent.add(llist);
Vector data = new Vector();
for (int i = 0; i < items.length; i++) {
    data.add(items[i] + (mod[i] ? " *" : ""));
}
JList list = new JList(data);
llist.setLabelFor(list);
parent.add(new JScrollPane(list));
```

Without augmentation this example generates a GUI like:

Figure 4 Swing GUI showing static text and an itimized list

And is represented internally as:

JPanel	panel	enabled,focus...					
JLabel	label	enabled,focus...					'Modified items are suffixed with "*"'
JScrollPane	scroll pane	enabled,focus...					
JViewport	viewport	enabled,focus...					
JList	list	enabled,focus...					

Figure 5 Tree showing default accessibility attributes

With augmentation as included in the code, this example generates a GUI like:

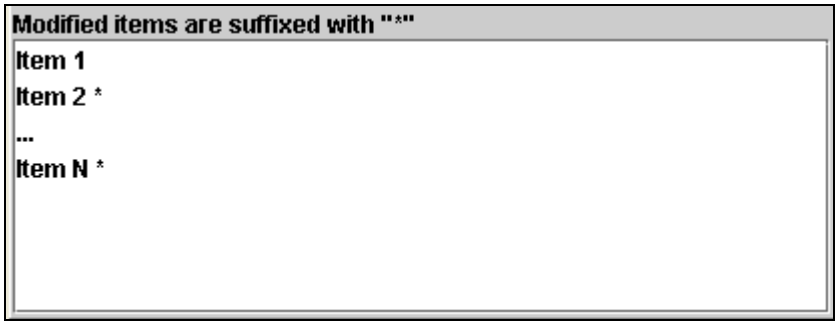

Figure 6 Swing GUI with added accessibility attributes

And is represented internally as:

JPanel	panel	enabled,focus...					
JLabel	label	enabled,focus...			label for:label 'Modifi...		'Modified items are suffixed with "*"'
JScrollPane	scroll pane	enabled,focus...					
JViewport	viewport	enabled,focus...					
JList	list	enabled,focus...					'Modified items are suffixed with "*"'

Figure 7 Tree showing added accessibility attributes

Note the effect of setting the "Label For" relationship on the `JLabel` and the addition of "*" as markers.

SWT Example:

```
// items in list and associated modified flags
String[] items = {"Item 1", "Item 2", …,  "Item N"};
boolean[] mod =  {false,    true,     …,  true};

// create list with modified indicators
Label llist = new Label(parent, SWT.LEFT);
llist.setText("Modified items are suffixed with \"*\"");
int style = SWT.SINGLE | SWT.H_SCROLL | SWT.V_SCROLL;
List list = new List(parent, style);
for (int i = 0; i < items.length; i++) {
     list.add(items[i] + (mod[i] ? " *" : ""));
}
```

The examples above add the "*" suffix to all modified items.

2.1.6 Give each Control a usable description

For people who are visually impaired, when a Control receives focus, an assistive technology must announce what the Control is and what type of information the user should provide by it or what action(s) the user should take with it. The assistive technology does this by presenting the Control's description to the user.

In accessibility there are several forms of description:

- **Role**—an indicator of the purpose and possibly the function of the Control. The role is typically presented by an assistive technology by default when the Control receives focus. Roles are a characteristic of the Control and should not be changed. Custom Controls can specify their roles.

- **Name**—a very brief description, typically a word or short phrase, of the Control. The name text is typically presented by an assistive technology by default when the Control receives focus. It can be its explicitly assigned or defaulted. You can provide the value used in code.

- **Short Description** (or Description)—a brief description. The description text is typically presented by an assistive technology only upon the user's request. You can provide the value used in code.

- **Long Description**—a complete description, typically a paragraph or more. It is generally presented by separate literal text in the GUI, by a GUI method such as a button launching a pop-up dialog or in help text. It is usually presented by the assistive technology as normal GUI content.

- **Value**—the actual value of the Control. For example, the text of a Text Control or the position of a Slider Control. Many Controls have no value. If it is small (a few words or numbers) the value is typically presented by an assistive technology by default when the Control receives focus. Larger values, such as a page of text, may require extra steps to be presented. It can be explicitly assigned or defaulted. You can provide the value used in code.

Each input Control should have a brief description of its function. This description supplements any description assigned to it by the GUI run time. Image-only Controls (such as a label with only an image) must have a description of the image's content. Often this is given in the form of a tool-tip. In Swing and SWT use the `setToolTipText` method to define the text.

For some Controls, such as buttons, the name is automatically set from the text of the Control. For many Controls the short description is set from any tool tip text defined for the Control. You can use the accessibility APIs to explicitly set different values for the name and description than provided by the Control itself. This can be helpful if, for example, you wish to have an assistive technology present different descriptive phasing than you want to use as a tool tip.

Note that this labeling needs to take into account language layout order (e.g., left-to-right vs. right-to-left). So a label might need to be placed visually after the Control it labels. This is easily accomplished in Swing with its explicit relationships, but may be more difficult in SWT with its order-of-creation-based relationships, especially with some `LayoutManagers`. Explicit physical placement of Controls may be needed (by using `Control.setBounds(…)` or `Control.setLocation(…)`.

Swing Example:

```
// use a tool tip as the description
JButton button = new JButton("Next");
button.setToolTipText("Press to advance to the next item");

// provide an override description
button.getAccessibleContext().setAccessibleDescription(
       "advance to the next item");
parent.add(button);
```

SWT Example:

```
// use a tool tip as the description
Button button = new Button(parent, SWT.PUSH);
button.setText("Next");
```

```
button.setToolTipText("Press to advance to the next item");

// provide an override description
button.getAccessible().addAccessibleListener(
                          new AccessibleAdapter() {
    public getDescription(AccessibleEvent e) {
        e.result = "advance to the next item";
    }
});
```

For each Control, you should test the name and descriptive phrasing presented by your assistive technology to be sure it is adequate for use of the Control. This is most important for Controls that contain images or figures. Consider a label showing a bar-chart. The name of the label might be "Bar Chart", the description might state the subject of the chart as "Sales per Region" and a long description might say "Sales per Region – North: $7,000, East: $10,000, South: $5,000 and West: $12,000." Note there is no need to describe the picture itself (e.g., describing the layout, colors or shapes).

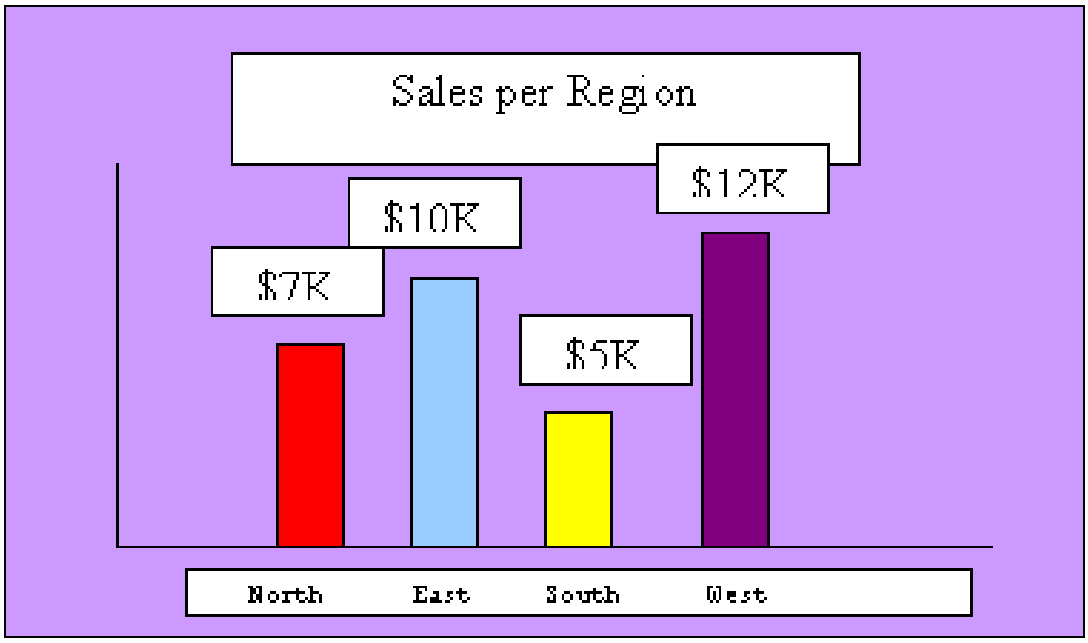

Figure 8 Sales per Region bar chart

2.1.7 Ensure you create accelerators for key Controls

Enabling your applications for use by a keyboard is a critical accessibility-enablement step (see Ensure a valid tab order is established). Navigation around your GUI via the keyboard must be possible and should be efficient and easy.

One way to achieve this is to add keyboard accelerators—sometimes called mnemonics—to critical Controls. This allows the user to navigate directly to the Control by use of the accelerator.

Some systems allow some Widgets, such as menu items, to have both mnemonics and accelerators. In this case, the mnemonic is used when the menu item is showing in its menu and the accelerator is used when the menu is not showing.

A key Control is a Control that is the first in a group of Controls or a stand-alone Control. Control groups are usually identified by having a titled border or other segregated space in the GUI. The following Controls should have accelerators (or mnemonics): menu items, push and radio buttons, check boxes, text fields or the first Control in a group.

Translation of accelerators and mnemonics gets complicated when there are lots of them in a set of menu items or a dialog. We suggest you restrict accelerator assignments to frequently used actions and not put them on every menu or dialog item. Some translations may produce duplicate character usage for different items, negating the function of the accelerator. When menus and dialogs are dynamically constructed, similar issues can occur, so it's important not to over use accelerators. Having too many counteracts their value. In general there are at most 26 accelerators (i.e., A-Z) available [7] and many languages have fewer then 26 letters at their disposal.

Since text fields do not support accelerators themselves [8] they need to be associated with a label that has the accelerator. In SWT that label needs to be created in its parent Composite immediately before the text field it labels. In Swing use the label's `setLabelFor` method to make the association. As described in Give each Control a usable description, text fields should also be labeled for descriptive reasons.

Most systems support Control keys for activating accelerators. If they do not and you have to code your own activation, make sure you do not reuse any system Control key sequence to activate an accelerator. If you support multiple platforms, make sure you avoid all the system's Control keys. In particular avoid "Alt" qualified keys. Do not make overly complex Control key sequences; try to restrict them to a two-key sequence.

As keyboard accessibility is critical, all accelerator and mnemonic key assignments must be documented in an easily accessible format. These

[7] Other keys, even special keys such as the function keys, may be used on some systems but we recommend against using them.

[8] In Swing, a text control can have an accelerator, but it has no way show a prompt for it (i.e., the user must remember the accelerator key), so we recommend you use an associated label with an accelerator instead.

assignments should be a primary help system topic. Also, any menu item should indicate the accelerator. For Swing and SWT this is automatic once you have defined the accelerator. In SWT, menu item mnemonics are set by preceding the mnemonic character with a "&." Accelerators are set using the `setAccelerator` method.

Swing Example:

```
// create a labeled text field with a mnemonic
JTextField field = new JTextField(20);
JLabel label = new JLabel("User name:");
label.setDisplayedMnemonic('N');
label.setLabelFor(field);
parent.add(label);
parent.add(field);
```

This example generates an unadorned GUI like:

Figure 9 Swing GUI showing a JLabel and JTextField

And is represented internally as:

⊟ JPanel	panel	enabled,focus...					
JLabel	label	enabled,focus...					'User name:'
JTextField	text	enabled,focus...		page...			

Figure 10 Tree showing the above components

Which generates the adorned GUI like:

Figure 11 Swing GUI showing a JLabel with a mnemonic labeling a JTextField

Note the underscore under the "n" in name indicating it is a mnemonic.

And is represented internally as:

JPanel	panel	enabled,focus...					
JLabel	label	enabled,focus...			label fo...		'User name:'
JTextField	text	enabled,focus...		page...			'User name:'

Figure 12 Tree showing the above components with extra attributes

Note the additional "Label For" relationship.

SWT Example:

```
// create a labeled text field with a mnemonic
Label label = new Label(parent, SWT.LEFT);
label.setText("User &name:");  // set mnemonic via '&'
Text field = new Text(parent, SWT.SINGLE);
```

Note that the order of creation of the label and text field is important in SWT but not in Swing as the "label-for" relationship is set explicitly (see Establish inter-Control relationships).

2.1.8 Establish inter-Control relationships

Often Controls in a GUI relate to other Controls in the GUI. Assistive technologies can use relationships to give a richer end-user experience.

As discussed in 2.1.7, you can label other Controls. When you place Controls in a group they become members of the group. SWT provides relationships implicitly, so creating the GUI creates the relationships. This behavior can be platform dependent. Swing also creates automatic relationships, but you can also manually add additional ones.

In Swing, we recommend that you establish all relationships explicitly, even if they are redundant, with an automatically-created relationship. Swing supports these relationships:

- Labeled by—this Control is labeled by another (a JLabel). More than one Control can be labeled by a single JLabel.
- Label for—this JLabel is the label of another Control. More than one Control can be labeled by a single JLabel.
- Controlled by—this Control is controlled by another Control.
- Member of—this Control is a member of a group defined by another Control.

The *Labeled* relationship can be defined:

Swing Example:

```
JLabel label = new JLabel("First Name:");
label.setName("firstNameLabel");
JTextField tf = new JTextField(40);
tf.setName("firstNameField");

// create a LABEL_FOR relationship in the JLabel that
// indicates that the label is a label for the text field.
AccessibleRelationSet arSet1 =
    label.getAccessibleContext().
          getAccessibleRelationSet();
AccessibleRelation ar1 = new AccessibleRelation(
                 AccessibleRelation.LABEL_FOR,
                 tf);
arSet1.add(ar1);


// create a LABEL_BY relationship in the JTextField that
// indicates that the text field is labeled by the
// label of First Name.
AccessibleRelationSet arSet2 =
    tf.getAccessibleContext().
          getAccessibleRelationSet();
AccessibleRelation ar2 = new AccessibleRelation(
                  AccessibleRelation.LABELED_BY,
                  label);
arSet2.add(ar2);
```

Note that the `JLabel.setLabelFor` method establishes the LABEL_FOR relationship but not the LABELLED_BY relationship. Explicitly setting both directions is recommended.

You can have one JLabel for multiple Controls. For example, this can be used in a group of radio buttons to label each button with a single label for the group, generating a GUI like:

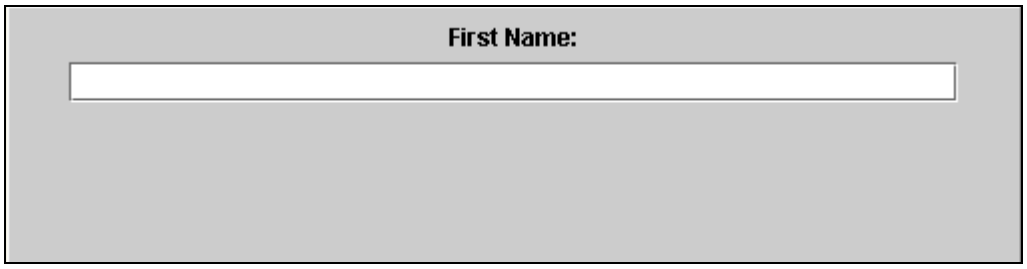

Figure 13 Swing GUI showing a JLabel and JTextField

This example is represented, without annotations, internally as:

JPanel	panel	enabled,focus...					
JLabel	label	enabled,focus...					'First Name:'
JTextField	text	enabled,focus...		page-...			

Figure 14 Tree showing the JLabel and JTextField

And is represented, with annotation, internally as:

JPanel	panel	enabled,focus...					
JLabel:firstNameLa...	label	enabled,focus...			label for:firstNameField		'First Name:'
JTextField:firstNam...	text	enabled,focus...		page...	labeled by:firstNameLabel		

Figure 15 Tress showing the JLabel and JTextField with extra attributes

Note the names on the components and the additional relationships.

The *Member Of* relationship can be defined:

```
// create a named group panel
JPanel group = new JPanel();
group.setBorder(new TitledBorder("Options"));
group.getAccessibleContext().
      setAccessibleName("Options");

// create each check box as a member of the group
JCheckBox cb1 = new JCheckBox("Case Sensitivity");
group.add(cb1);
AccessibleRelationSet arSet1 =
```

```
    cb1.getAccessibleContext().
        getAccessibleRelationSet();
AccessibleRelation ar1 = new AccessibleRelation(
                  AccessibleRelation.MEMBER_OF,
                  group);
arSet1.add(ar1);

JCheckBox cb2 = new JCheckBox("Regular Expression");
group.add(cb2);
AccessibleRelationSet arSet2 =
    cb2.getAccessibleContext().
        getAccessibleRelationSet();
AccessibleRelation ar2 = new AccessibleRelation(
                  AccessibleRelation.MEMBER_OF,
                  group);
arSet2.add(ar2);

JCheckBox cb3 = new JCheckBox("Wrap Beginning/End");
group.add(cb3);
AccessibleRelationSet arSet3 =
    cb3.getAccessibleContext().
        getAccessibleRelationSet();
AccessibleRelation ar3 = new AccessibleRelation(
                  AccessibleRelation.MEMBER_OF,
                  group);
arSet3.add(ar3);
```

Which generates a GUI like:

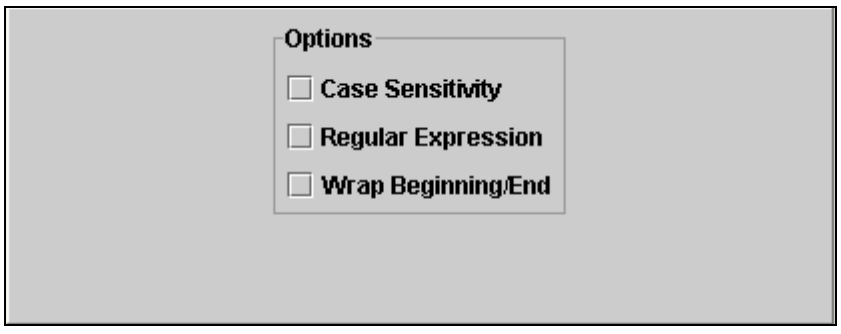

Figure 16 Swing GUI showing three JCheckBoxes in a group

And is represented, without annotation, internally as:

JPanel	panel	enabled,focus...					
JPanel	panel	enabled,focus...					'Options'
JCheckBox	check box	enabled,focus...	0	click			'Case Sensitivity'
JCheckBox	check box	enabled,focus...	0	click			'Regular Expression'
JCheckBox	check box	enabled,focus...	0	click			'Wrap Beginning/End'

Figure 17 Tree showing the JCheckBoxes

And is represented, with annotation, internally as:

JPanel	panel	enabled,focus...					
JPanel	panel	enabled,focus...					'Options'
JCheckBox	check box	enabled,focus...	0	click	member of:panel 'Options'		'Case Sensitivity'
JCheckBox	check box	enabled,focus...	0	click	member of:panel 'Options'		'Regular Expression'
JCheckBox	check box	enabled,focus...	0	click	member of:panel 'Options'		'Wrap Beginning/End'

Figure 18 Tress showing the JCheckboxes now as members of the Options group

Notice the additions of the ",member of" relationship for each check box.

The examples above are based on those in the *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java*.

2.2 Event-time Guidelines

Apply the checks in this section for each event callback handler you create.

2.2.1 Avoid automatic advancement

People have varying ability to react to prompts; persons with disabilities may require extended periods of time to respond.

Do not require input in a fixed amount of time. For example, do not proceed to the next step in a sequence after a fixed delay. If you must support automatic advancement, make the delay configurable, or make it very easy to extend the delay time. The default delay (before configuration) should be infinite.

Swing Example:

```
// create a button with auto-press
final JButton button = new JButton("Press Me!");
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        autoPush.stop();   // prevent auto action
        // do action…
    }
});
```

```
parent.add(button);
:
// get the delay value to use from the system;
// this makes the delay configurable
Integer delay = Integer.getInteger("button.auto.delay",
                                   Integer.MAX_INT);
final Timer autoPush =
  new Timer(delay.intValue(), new actionListener() {
    public void actionPerformed(ActionEvent ae) {
        button.doClick();   // cause auto action
    }
});
autoPush.setRepeats(false);

// allow delay to be extended by pressing any key
parent.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent ke) {
        if(autoPush.isRunning()) {
            autoPush.restart();
        }
    }
});
autoPush.start();.
:
```

SWT buttons do not support programmatic activation. JFace application may add this capability through the *IAction* interface

2.2.2 Ensure your GUI is responsive

Users expect a GUI to respond to their input, so a GUI should never appear to "hang". If delays are unavoidable, an indication that the delay is in progress is required. This can require careful coding on your part. The basic rule is to never do any action on a GUI event callback beyond updating the GUI itself. All event handlers must be checked to ensure that they do not cause GUI processing to be delayed.

This means doing anything that can block (even if it typically does not block) or takes more than a few tenths of seconds to complete should be done on a separate *thread*. Some examples are reading from a file, sending a message on the network or a long computation. All threads take a `Runnable` object that contains a `run()` method. You place the steps of your long-running process in the run method. Eclipse offers an enhanced version of this interface that passes in an `IProgressMonitor` object so that visual progress updates can be easily shown to the user.

Java provides threads to support asynchronous activity. Threads are lightweight processes running within a single JVM. Both Swing and SWT use a thread to dispatch GUI events (e.g., mouse or key presses). This thread dispatches one event at a time. There is an *event queue* used to buffer events before they can be processed. Both Swing and SWT are non-reentrant, meaning that only one thread can be running in SWT or Swing code at any time. To prevent any chance of reentrancy, both Swing and SWT specify that all GUI updates—such as setting a label's text or any Controls colors—must be done on the event thread. Swing provides the `SwingUtilities.invokeAndWait(Runnable)` or `SwingUtilities.invokeLater(Runnable)` methods to enforce this. SWT has similar methods called `Display.syncExec(Runnable)` and `Display.asyncExec(Runnable)`. SWT will throw an exception if you violate this rule. Swing does not but it may give unexpected and undesirable results if you violate this restriction. If you update your GUI inside a (non-GUI) thread's run method, you must use these utilities to wrap code that updates your GUI. You do not need to use these utilities if you are on the GUI thread (such as in a Button's `ActionListener`).

Executing actions asynchronously further means you need to disable some capabilities of your GUI. For example, unless you have support to queue requests, you need to disable the"Save" button when a save operation is in progress.

For some delays it may be sufficient to present a wait cursor instead of using a separate thread. This technique is not recommended if the delay can ever exceed a few seconds.

Swing Example:

```
// fields
final JTextField filenameField = new JTextField(40);
final JButton saveButton = new JButton("Save");

: // in GUI construction code

// button press event handler
saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        saveButton.setEnabled(false);
        Thread t = new Thread(new Runnable() {
             public void run() {
                 doSave(filenameField.getText());
             }
         });
         t.start();
```

```
    }
});
parent.add(filenameField);
parent.add(saveButton);
:
// process a save request
public void doSave(String filename) {
    try {
        // … save the file
    }
    catch (IOException ioe) {
        // … do some recovery and/or reporting action
    }
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            JOptionPane.showMessageDialog(parent,
              "Save complete", "Save",
              JOptionPane.INFORMATION_MESSAGE);
            saveButton.setEnabled(true);
        }
    });
}
```

-- or --

```
// fields
protected JTextField filenameField = new JTextField(40);
protected JButton saveButton = new JButton("Save");
protected JFrame application = …

: // in GUI construction code

// button press event handler
saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        Cursor activeCursor = application.getCursor();
        application.setCursor(
           Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
        doSave(filenameField.getText());     }
        application.setCursor(activeCursor);
    });
parent.add(filenameField);
parent.add(saveButton);
:
// process a save request
public void doSave(String filename) {
    try {
```

```
        // … save the file
    }
    catch (IOException ioe) {
        // … do some recovery and/or reporting action
    }
}
```

SWT Example:

```
// fields
protected Display display;
protected Text filenameField;
protected Button saveButton;

: // in GUI construction code

display = parent.getDisplay();
filenameField = new Text(parent, SWT.SINGLE);
saveButton = new Button(parent, SWT.PUSH);
button.setText("Save");

// button press event handler
saveButton.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent se) {
        saveButton.setEnabled(false);
        Thread t = new Thread(new Runnable() {
             public void run() {
                 doSave(filenameField.getText());
             }
         });
         t.start();
    }
});
:
// process a save request
public void doSave(String filename) {
    try {
        // … save the file
    }
    catch (IOException ioe) {
        // … do some recovery and/or reporting action
    }
    display.asyncExec(new Runnable() {
        public void run () {
             MessageDialog.openInformation(parent,
                         "Save", "Save complete");
             saveButton.setEnabled(true);
```

```
        }
    });
}
```

-- or --

```
// fields
protected Display display;
protected Text filenameField;
protected Button saveButton;
protected Composite application;

: // in GUI construction code

application = …;
filenameField = new Text(parent, SWT.SINGLE);
saveButton = new Button(parent, SWT.PUSH);
saveButton.setText("Save");

// button press event handler
saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        application.setCursor(
           display.getSystemCursor(SWT.CURSOR_WAIT));
        doSave(filenameField.getText());
        application.setCursor(null);
    }
});
:
// process a save request
public void doSave(String filename) {
    try {
        // … save the file
    }
    catch (IOException ioe) {
        // … do some recovery and/or reporting action
    }
}
```

SWT provides the `BusyIndicator` class to make this easier. The above example can be rewritten using `BusyIndicator` as:

```
: // in GUI construction code
filenameField = new Text(parent, SWT.SINGLE);
saveButton = new Button(parent, SWT.PUSH);
saveButton.setText("Save");
```

```
// button press event handler
saveButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        BusyIndicator.showWhile(display, new Runnable() {
            public void run() {
                doSave(filenameField.getText());
            }
        });
    }
});
:
```

Eclipse provides a variety of methods to ensure responsive GUIs. For longer running processes it provides several different forms of progress monitors. The status line contains a progress bar you can use. For example, from any view you can show progress as follows:

```
IActionBars ab = getViewSite().getActionBars();
IStatusLineManager slm = ab.getStatusLineManager();
IProgressMonitor pm = slm.getProgressMonitor();
int steps = …;
pm.beginTask("Doing a " + steps + " step process now",
             steps);
for(int i = 0; i < steps; i++) {
    // … do next step …
    pm.worked(1);  // advance work done
}
pm.done();
```

Eclipse 3.0 provides a progress service that implements smart wait notification. It uses a busy cursor for short waits and then changes to a progress bar notifier if the delay exceeds a predetermined threshold (set by preferences and accessed by `IProgressService.getLongOperationTime()`). For example:

```
IWorkbench wb = PlatformUI.getWorkbench();
IProgressService ps = wb.getProgressService();
ps.busyCursorWhile(new IRunnableWithProgress() {
    public void run(IProgressMonitor pm) {
        // … do some long running task here …
    }
});
```

Eclipse 3.0 has created the notion of "jobs" which are potentially long-running tasks that the user may want to monitor and Control through a Control panel. Jobs can be grouped into "families" that can be manipulated as a group. Jobs can be run real time or scheduled for execution at a later time. Jobs can repeat.

Jobs have names and priority and other user assigned properties. Jobs can have "rules" that determine when they run. It is possible to run jobs in a dialog using a progress service.

```
IWorkbench wb = PlatformUI.getWorkbench();
IProgressService ps = wb.getProgressService();

Job myJob = new Job("My Job") {
    private volatile boolean cancelled;
    public IStatus run(IProgressMonitor pm) {
        boolean done = false;
        while(!cancelled && !done) {
            // … do next step of job task here …
            pm.worked(1);
            done = … some condition test … ;
        }
        return cancelled ? Status.CANCEL_STATUS
                         : Status.OK_STATUS;
    }
    public boolean cancel() {
        cancelled = true;
        return super.cancel();
    }
});
ps.showInDialog(shell, myJob);
myJob.schedule();
```

Note: in all these long-running options if you update the GUI you must do it in a `Display.asyncExec(…)` method.

The dialogs shown above are *modal*—they stop interaction with the GUI until they are dismissed. Being modal can have a negative impact on the responsiveness of the GUI. For example, it is generally preferable for a "find" or "help" dialog not to be modal so that you can use it and still use the rest of the GUI. SWT supports four modalities: non-modal, APPLICATION_MODAL, PRIMARY_MODAL and SYSTEM_MODAL. Each modal in SWT is specified by a style bit, while Swing supports just modal (like APPLICATION_MODAL) and non-modal.

2.3 Design Guidelines

Apply the checks in this section for each distinct GUI you create. As these guidelines are best implemented at design time, specific code inspection acts are premature. These guidelines are more like rules-of-thumb to follow to validate the completeness of a function line-item list or product-component design.

2.3.1 Ensure you can adjust attributes of your GUI

Not everyone perceives information the same way. This is particularly true for persons with visual or auditory impairments.

Each user must be able to adjust the presentation of information to match his or her needs. For example, the volume of a sound and the color of the text, the size of the text or both must be adjustable.

Most operating systems have the means to make global changes such as system volume and GUI Control suites colors and sizes. Where ever possible you should leverage these features and your code should not interfere with them. Avoid using custom colors, fonts or font sizes. This means any use of `set…Color…`, `set…Font..`, etc., methods should be carefully examined for necessity.

If you must use your own colors, ensure that the colors provide sufficient contrast so that adjacent content can be easily distinguished. Take care to make the difference between the foreground and background colors significant.

Also note that using color for contrast is an issue when printing to monochrome devices such as black and white (B&W) printers. Often the content becomes unreadable due to lack of contrast in the colors. So printing your GUI content on a B&W printer can be a quick (but not always sufficient) test to see if you have good contract.

Avoid using images to present textual information. In an attempt to make more dramatic GUIs it has become common to render text content in stylized fashion through images. If you must do this, your images should have descriptions (see Give each Control a usable description) that provide the text included in the image. Consider instead using the Swing *JEditorPane/JTextPane* or SWT *StyledText* Controls to present formatted text.

Similar issues occur with patterned backgrounds. If you use these they must be able to be disabled or redefined to meet a user's visual needs.

For extra usability, you can provide a method to easily scale (or zoom) your GUI. This allows users to quickly adjust the size of the GUI to meet their needs. You can support independent font size selection, GUI-wide zoom Control or both.

Note that scaling an entire GUI application (e.g., from the frame/shell on down) is difficult. There is no foolproof way [9] to scale the entire application in one step.

[9] It is possible to scale an entire application outside of Swing using host-specific functionality. This is typically what a "screen magnifier" assistive technology will do.

Each Control in the GUI may need to be scaled individually, although you may be able to scale a Composite and have its children automatically scaled as well, depending on the child Control's drawing behavior.

Scaling for an individual Control is best done by subclassing it, overriding the `paint(Graphics)` or `paintChilren(Graphics)` methods and adjusting the *Graphics* (really a *Graphics2D*) object to scale the content. This works for Swing only; SWT does not offer this support. To do similar things in Eclipse [10] you must also have *Draw2D* support installed from the Eclipse Graphical Editor Framework.

If you must use your own colors or fonts, you are then required to provide ways to adjust them. This can mean a new set of user dialogs and design changes. You may need to provide alternates for some elements of your GUI.

Swing Example:

```
/** A JPanel that scales its children */
public class ScalePanel extends JPanel {
    protected double scale = 1.0;
    public double getScale() {
        return scale;
    }
    public void setScale(double scale) {
        this.scale = scale;
    }

    public ScalePanel(double scale) {
        setScale(scale);
    }

    public void paintChildren(Graphics g) {
        if (scale != 1.0) {
            Graphics2D g2 = (Graphics2D)g.create();
            try {
                // adjust the Graphics to the new scale
                g2.clipRect(0, 0, getWidth(), getHeight());
                g2.scale(scale, scale);
                // paint my children at the new scale
                super.paintChildren(g2);
            }
            finally {
                g2.dispose();
            }
        }
```

[10] Details on how to do this are beyond the scope of this document.

```
        else {
            super.paintChildren(g);
        }
    }
}
```

2.3.2 Use multi-modal indicators

Some users may have auditory disabilities or may work in noisy environments. They also may not be able to give full concentration to the GUI, missing transitory GUI content. Often assistive technologies cannot present alternate presentation of auditory or transitory information.

Avoid the use of sound as the only indicator. In particular do not use sound as the only indicator that it is time to proceed to a next step in a sequence or to take some corrective recovery action. Provide at least one visual indicator as well.

Do not relay on only intermittent visual indicators. The visual indicator needs to be in the main area of the GUI (e.g., in the normal tab order flow). A simple change to a status-line field, for example, may be missed by an assistive technology if focus does not flow to the status line. In some cases, a pop-up dialog (such as Swing's `JOptionPane` or SWT's `MessageDialog`) may be needed to signal the event. See the example in Ensure your GUI is responsive.

Avoid the use of flashing Controls or text. Flashing is often disturbing and can trigger serious adverse reactions in some people. If you must use flash, do not use the 2-55Hz frequency range and avoid large flashing areas or flashing with strong color contrast. A flashing caret for text entry is acceptable.

2.3.3 Ensure you provide accessible content

Modern GUIs no longer consist of just input fields and buttons. Today, much more sophisticated input and output Controls are being used. In particular multi-media presentation is becoming very popular. Without accessible content assistive technologies cannot present these complex GUIs.

Each time you create a multi-media Control (such as a music or DVD player) you must ensure that all of the content for it can be presented to all users. For example, if you have a visually-animated tool to instruct the user how to perform a task, there must be a way to provide the same information in a nonanimated (such as single frame at a time mode with each frame having a description) or a nonvisual format (such as in text and/or an audio description). Any speech presented in the animation must also be available in other ways. For example, the spoken words should be provided in text form as a (closed-) caption.

As a developer, you need to ensure that all alternate presentation forms and mechanisms are available in your GUI. This can require design and coding changes. If you discover that some content is not available in all required forms, you should request that it be obtained. We recommend that you make the selection of presentation of alternate content forms easily configurable.

You are not required to do this is if the user selects the content (such as if you are providing a music or video player), only if it is content you provided as part of your application (e.g., a tutorial). Still, if the user selects content that has alternate formats (e.g., closed-caption tracks), you should have a means to present them.

When selecting multimedia Controls, whenever possible make sure that you select ones that provide the necessary support for alternate content streams.

3 Coding Guidelines Evaluation Checklist

Module:

These checks are needed each time you create a Control:

Relevant	Verified	Guideline
☐	☐	2.1.1 Ensure a valid tab order is established.
☐	☐	2.1.2 Ensure some control gets focus.
☐	☐	2.1.3 Give each Control a unique identifier.
☐	☐	2.1.4 Ensure all text is seen.
☐	☐	2.1.5 Use text attributes only for embellishment, not information.
☐	☐	2.1.6 Give each Control a usable description.
☐	☐	2.1.7 Ensure you create accelerators for key Controls.
☐	☐	2.1.8 Establish inter-Control relationships.

These checks are needed for event handlers:

Relevant	Verified	Guideline
☐	☐	2.2.1 Avoid automatic advancement.
☐	☐	2.2.2 Ensure your GUI is responsive.

These checks may be needed when you design your GUI:

Relevant	Verified	Guideline
☐	☐	2.3.1 Ensure you can adjust attributes of your GUI.
☐	☐	2.3.2 Use multi-modal indicators.
☐	☐	2.3.3 Ensure you provide accessible content.

4 Resources

The following documents and tutorials provide additional information about enabling GUIs for accessibility.

- *Software Accessibility* checklist - http://w3-03.ibm.com/able/devtest/software.html
-
- *IBM Guidelines for Writing Accessible Applications Using 100% Pure Java* - http://w3-03.ibm.com/able/devtest/software.html#resources
-
- *Using JAWS to verify accessibility* - http://w3.austin.ibm.com/~snsinfo/-devtest/jawsref.html
- *Test Resources* - http://w3-03.ibm.com/able/devtest/AC_Test_Resources.html

.