Programming with Java™ AWT

Rex Jaeschke

Programming with Java AWT

Edition: 4.0 (matches JDK1.6/Java 2)

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Preface

This book covers the Java Abstract Windows Toolkit (AWT) and its use in implementing Graphical User Interfaces (GUIs) in Java applications and, to some extent, Java applets. This course is based on the features made available starting with JDK V1.1.

Reader Assumptions

This text follows on from its companions *Programming in Java* and *Java Applets*. You are expected to be comfortable with the following concepts and the syntax required to express them in Java:

- Basic Language Elements
- Looping and Testing
- Methods
- References, Strings, and Arrays
- Classes
- Inheritance
- Exception handling
- Input and Output
- Packages
- Interfaces

Source Code, Exercises, and Solutions

The programs shown in the text are made available electronically in a directory tree named Source, where each chapter—as well as each example within that chapter—has its own subdirectory.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided on disk in a directory tree named Labs, in which each chapter has its own subdirectory.[1] Exercises that are not so marked have no general solution and require experimentation or research in an implementation's documentation. Numerous exercises contain a statement of the form "(See directory *xx*.)". This indicates the corresponding solution or test file in the Labs subdirectory.

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

Screen Displays

The windows pictured throughout this text were produced on one or more versions of Microsoft Windows. Since different windowing systems present menus and information in different ways, you will need to extrapolate when using a different system.

What's Not Covered

The GUI support introduced with the Swing tools of the Java Foundation Classes (JFC) is covered in a separate course.

JDK1.2 introduced the following new AWT subpackages, none of which is covered in this book:

- `java.awt.color` — Provides classes for color spaces.
- `java.awt.datatransfer` — Provides interfaces and classes for transferring data between and within applications.
- `java.awt.dnd` — Provides interfaces and classes for supporting drag-and-drop operations.
- `java.awt.font` — Provides classes and interface relating to fonts.
- `java.awt.geom` — Provides Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
- `java.awt.im` — Provides classes and an interface for the input method framework.
- `java.awt.image` — Provides classes for creating and modifying images.
- `java.awt.image.renderable` — Provides classes and interfaces for producing rendering-independent images.
- `java.awt.print` — Provides classes and interfaces for a general printing API.

The Java Development Kit

Sun's initial production release of Java was the Java Development Kit (JDK) version 1.0. Versions 1.1 through 1.6 contained numerous bug fixes, and language and library enhancements. The latest version can be downloaded from Sun's website (www.sun.com).

While the language has remained very stable, along the way, several features were added along with numerous new packages and classes and new methods to existing classes. Also, some existing method names have been changed. In these latter cases, the old names continue to be acceptable, but are flagged by the compiler as *deprecated*, meaning that support for them might well be removed in future versions. If your compiler issues such a warning, consult Sun's on-line documentation to find the recommended replacement.

From an internationalization (I18N) viewpoint, one of the most significant additions made by V1.1 was the completion of support for dealing with non-US, non-English environments, including those involving very large alphabets and non-Latin writing systems.

Acknowledgments

Thanks to those people who reviewed all or part of this book. In particular, students in my seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, October 2009

1. AWT Basics

In this chapter, we will be introduced to the Abstract Windows Toolkit (AWT) package and some basic Graphical User Interface (GUI) design ideas. We will also learn about the classes `Component`, `Container`, `Dialog`, `Frame`, `Panel`, and `Window`.

1.1 Introduction

The Abstract Windows Toolkit package contains numerous classes that can be used to create Graphical User Interfaces (GUIs). A GUI is made up of one or more objects such as buttons, checkboxes, or scrollbars, each of which is called a *component*. When we interact with a component by pressing its button, checking its checkbox, or moving its scrollbar, for example, that component generates an *event*, which can be detected and serviced by an *event handler*.[1] All components are rectangular in shape and have a horizontal or vertical orientation.

While a component is often a GUI object, some components contain other components in which case, they are referred to as *containers*. A container can contain components as well as other containers. A *top-level container* cannot be part of any other container. Every container is a component but a component is not necessarily a container. When a container contains multiple components, a *layout manager*[2] can be used to organize the visual placement of those components.

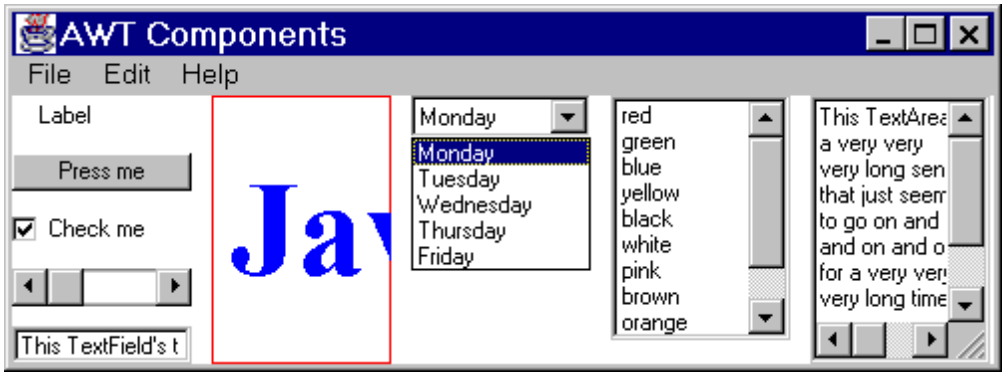
Here are the component classes that, except for `CheckBoxGroup`, are subclasses of `Component` or `MenuComponent`. (Examples of some of these components are shown on the following page.)

- `Button`: A button is pressed to cause the action indicated by its label.
- `Canvas`: A canvas is used as a drawing area for graphics.
- `Checkbox`: A checkbox is used to maintain a two-state condition; either the box is checked or it isn't.
- `CheckboxGroup`: This is a group of checkboxes, only one of which can be checked at a time.
- `Choice`: A choice consists of a drop-down list of items. Once a selection has been made from this list, only that selection is displayed.
- `Dialog`: A dialog is a top-level window that has a title bar and a border. It is like a frame but has fewer properties. A dialog is a container.
- `Frame`: A frame is a window that has a title bar, a menu bar, a border, a cursor, and an associated icon. A frame is a container.
- `FileDialog`: A file dialog allows a file to be specified or selected for load-from-disk or save-to-disk operations. A file dialog is a container.
- `Label`: A label displays a single line of text that can be modified by the program but not by the user. A label is generally used to label components that do not otherwise have their own label, such as text fields and text areas.
- `List`: A list contains a scrollable list of items.

[1] Events and event handling are discussed in §1.
[2] Component layout is discussed in §1.

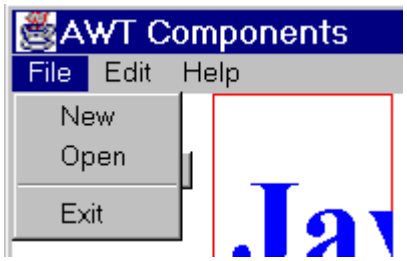
- `MenuBar`: A menu bar contains one or more Menus, each of which contains one or more MenuItems. MenuItems can be checked.
- `Panel`: A panel contains other components. Components are laid out on panels, which, in turn, can be laid out on other containers such as panels or frames.
- `PopupMenu`: A free-floating menu that "pops up", typically, as a result of right clicking. (Introduced in JDK1.1.)
- `Scrollbar`: A scrollbar contains a scroll box, which can be scrolled up and down. A scrollbar can have either a horizontal or a vertical orientation.
- `ScrollPane`: A viewing area that can be scrolled in either of two directions. (Introduced in JDK1.1.)
- `TextArea`: A text area displays one or more lines of text that can be edited. Horizontal and/or vertical scrollbars are used as necessary.
- `TextField`: A text field displays one line of text that can be edited. Although a scrollbar is not used, text can be scrolled by using various keys.
- `Window`: A top-level window that has no title bar, menu bar, or border. A window is a container.

Most of the components listed above are shown in the following GUI, the source code for which is contained in Ba01.java:



The components on the left are, from top-to-bottom, a label, a button, a checkbox, a horizontal scrollbar, and a text field; they are contained in a panel. To their right is a canvas bordered by a red rectangle and containing blue text. Then comes a list with drop-down list shown and the item Monday selected. In the fourth column, we have a choice with vertical scrollbar. Lastly, we have a text area with both scrollbars active.

The panel and the four components to its right are contained inside a frame, which has a menu bar. This bar has three Menus: File, Edit, and Help. When the File menu is selected, we see the following:



The other two Menus have no items.

Here is the hierarchy of the component classes:

```
Borderlayout
Cardlayout
CheckboxGroup
Color
   SystemColor
Component
   Button
   Canvas
   Checkbox
   Choice
   Container
      Panel
      ScrollPane
         Applet
      Window
         Dialog
            FileDialog
         Frame
   Label
   List
   Scrollbar
   TextComponent
      TextArea
      TextField
Cursor
Dimension
Event
EventQueue
FlowLayout
Font
FontMetrics
Graphics
GridBagConstraints
GridBagLayout
GridLayout
Image
Insets
MediaTracker
MenuComponent
   MenuBar
   MenuItem
      CheckboxMenuItem
      Menu
         PopupMenu
MenuShortcut
Point
Polygon
PrintJob
Rectangle
Toolkit
```

Exercise 1-1: Build and run Ba01.java. Interact with all of the components that you can. In particular, try the following:

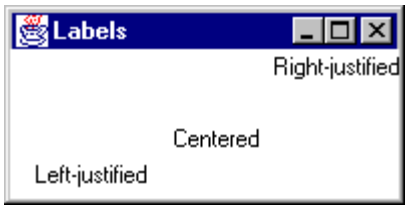
1. Press the button.
2. Check and uncheck the checkbox.
3. Scroll the scroll box in the horizontal scrollbar. Click on the up and down buttons.
4. Scroll through the text field by using the left- and right-arrow keys. Also, try using the Home and End keys. Delete text using the Backspace or Delete keys, mark a piece of text by clicking and dragging or by double-clicking, delete, or replace a marked piece of text, and insert text.
5. Activate the choice items, scroll through the choices, and select one.

4. Components

In this chapter, we will learn how to use the individual components that are not also containers.

4.1 Labels

A *label* is comprised of a single line of text that can be modified by the program at any time; it cannot be modified by the user, however, and so it cannot get the focus. However, it can detect mouse events. A label is commonly used to provide annotation for other components. The label's text can be horizontally aligned to the left, to the right, or centered. It is always centered vertically. Consider the following display, produced by Cmlb01.java, which contains four labels:



Labels 1, 3, and 4 are aligned as indicated by their text descriptions. The text of label 2 is empty, in which case, alignment is ignored. Here's the source for that program:

```
import java.awt.*;

class Cmlb01 extends Frame
{
    public Cmlb01()
    {
          super("Labels");
          setLayout(new GridLayout(4, 0));

/*1*/     Label l = new Label();
/*2*/     add(l);
/*3*/     l.setText("Right-justified");
/*4*/     l.setAlignment(Label.RIGHT);
/*5*/     add(new Label());
/*6*/     add(new Label("Centered", Label.CENTER));
/*7*/     add(new Label("Left-justified"));

          addWindowListener(new DetectWindowClose());
          setSize(200, 100);
          show();
    }
```

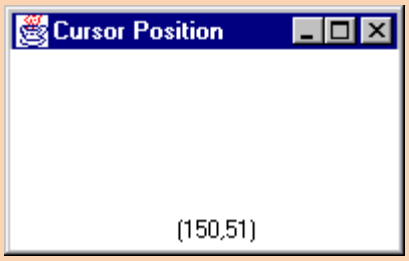
```
    public static void main(String[] args)
    {
          new Cmlb01();
    }
}
```

In case 1, we construct a label whose text is an empty string, which is left aligned by default. We then add that label to the frame in case 2. Method `Label.setText` can be called at any time to change the label's text. (A sibling method, `Label.getText`, returns the text as a string.) Changing a label's text can change the size of the component, depending on the layout being used. As such, it may be necessary to invalidate and revalidate. `Label.setAlignment` sets the alignment to `CENTER`, `LEFT`, or `RIGHT`, as specified, while `Label.getAlignment` returns one of these values as an `int`.

The new label created and added in case 5, has no text and is aligned left. That created in case 6 has the given text and is centered, while in case 7, we create a left-aligned label. Since no positions are specified, the labels are added to the grid layout in sequential order.

Exercise 4-1: Inspect the documentation for class `Label` to find out the available public constructors, methods, and fields.

Exercise 4-2*: Consider the following display:



When the mouse is moved or dragged over the frame, the current cursor position is displayed centered at the bottom. Write a program that reproduces this behavior. What happens when you move the mouse outside the frame? What about dragging it outside the frame? Make sure that the cursor position stays at the bottom center when the frame is stretched or compressed. (See lab file Lbcmlb01.java.)

4.2 Text Fields

A *text field* displays a single line of text; it is commonly used to solicit user input. Unlike a label, a user can edit the text in a text field. When a certain key (usually the carriage return) is pressed inside a text field, an action event is generated and the string representing the contents of the text field is made available.

The following display, produced by Cmtf01.java, solicits input from the user. When the Enter/Return key is pressed, both upper- and lowercase versions of the input string are displayed in separate text fields whose contents cannot be edited by the user.

Here are the relevant parts of Cmtf01.java:

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

class Cmtf01 extends Frame implements ActionListener
{
/*1a*/    private TextField input       = new TextField(15);
/*1b*/    private TextField inputUpcase = new TextField(15);
/*1c*/    private TextField inputLocase = new TextField(15);
```

In cases 1a–1c, we create three text fields, each having a display width of 15 characters and no initial contents. The width is based on the current font of that component, and limits only the display width. A text field can contain an arbitrarily long amount of text, which simply scrolls left or right as we enter more text or position using the Home, End, or direction keys. The actual width of a text field can be larger than that specified if that component is in a layout (such as GridLayout) that permits it to be stretched. We can find out a text field's width by calling `TextField.getColumns`; however, the value returned reflects the width specified when that component was constructed, not its current display width.

```
    public Cmtf01()
    {
          super("Input Case Conversion");

          setLayout(new GridLayout(0, 2, 10, 5));
          add(new Label("Input", Label.RIGHT));
          add(input);
          add(new Label("Uppercase Input", Label.RIGHT));
/*2*/     inputUpcase.setEditable(false);
          add(inputUpcase);
          add(new Label("Lowercase Input", Label.RIGHT));
          inputLocase.setEditable(false);
          add(inputLocase);

          input.addActionListener(this);
          addWindowListener(new DetectWindowClose());
          setSize(250, 100);
          show();
    }
```

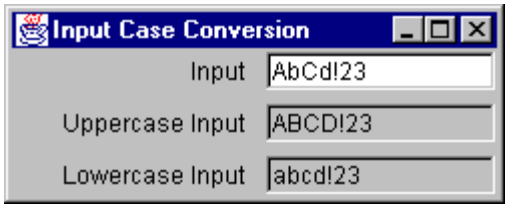
By default, the contents of a text field can be modified by the user; that is, a text field is *editable*. We can disable this property by calling `TextComponent.setEditable` with an argument of `false`, as in case 2. (Calling it with an argument of `true` makes it editable.) We can also determine of a text field is editable by calling `TextComponent.isEditable`.

```
    public static void main(String[] args)
    {
          new Cmtf01();
    }
    public void actionPerformed(ActionEvent e)
    {
/*3*/     String text = input.getText();

/*4a*/    inputUpcase.setText(text.toUpperCase());
/*4b*/    inputLocase.setText(text.toLowerCase());
    }
}
```

When Enter/Return is pressed in an editable text field, an event of type `ActionEvent` is generated allowing us to retrieve that text field's contents using `TextComponent.getText`, as in case 3. We can set the text in a text field at any time by calling `TextComponent.setText`, even if that text field is not editable.

Some implementations of the AWT provide a way to indicate whether a text field is editable. For example, the following display, produced by a different implementation than the one above, grays out text fields that are un-editable:



Class `TextField` has a sibling class, `TextArea`. Since these both display editable text, they have a common superclass, `TextComponent`, which provides common functionality. For example, the methods `getText`, `setText`, `isEditable`, and `setEditable` really belong to `TextComponent`.

For input like passwords, it is useful to be able to suppress the input as it is typed. Rather than actually suppress the text, we can cause some other character, called the *echo character*, to be displayed in place of each character input; that way we can see how many characters we've entered. Commonly used echo characters are `'*'` and `'?'`. We establish the echo character by calling `TextField.setEchoChar`, passing it a `char` argument, where a character with value 0 is reserved to mean "re-enable echoing of the input." By default, input is displayed when a text field is constructed. We can obtain the current echo character at any time by calling `TextField.getEchoChar` and we can determine if echoing has been enabled by calling `TextField.echoCharIsSet`.

The text contained in a TextComponent (and therefore, in a text field) has characters numbered starting from position 0. A newline occupies one position. A range of positions includes the start position and one more than the end position; for example, the range 0–4 includes 4 characters, at positions 0, 1, 2, and 3. We can select all or any one contiguous part of the text in a text field by calling `TextComponent.selectAll` or `TextComponent.select`, respectively. And most windowing systems provide a way to select text using the mouse.[1] Once a range has been selected, `TextComponent.getSelectionStart` and `TextComponent.getSelectionEnd` return the start and end position, respectively. `TextComponent.getSelectedText` retrieves the selected text. We can use these text selection methods to implement sophisticated text search and replacement operations.

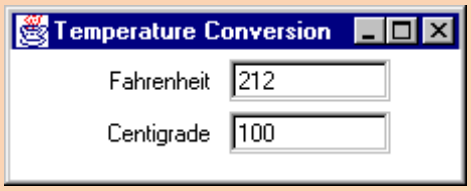
As well as selecting and manipulating text under program control, windowing systems typically provide a way to do this using the mouse.[2] Using this approach, we can copy and/or cut-and-paste from one text-oriented component to another.

Exercise 4-3: Inspect the documentation for class `TextField` and `TextComponent` to find out the available public constructors, methods, and fields.

Exercise 4-4*: Write a temperature conversion program that converts from Fahrenheit to Centigrade, and vice-versa. The conversion formula to use is as follows:

F = (C * 9 / 5) + 32)

Here's an example of what the display should look like:



When you enter a value in one text field, the converted result is displayed in the other text field. An input need not be a whole number. If an input string cannot be converted to a number, display “Invalid input” in the other text field. (See lab file Lbcmtf01.java.)

Exercise 4-5: In data entry, the best approach to validation is to reject an invalid character immediately it is input, so the user never has to re-input more than the offending character. A text field will accept any and all input, however, in many applications we'll likely wish to limit the kind of characters that we wish to accept for certain fields. For example, a cost field should accept fractional values, possibly with a maximum of two decimal places and optionally with thousand separators. We might also want to accept alphabetic input only and convert to either all upper- or all lowercase. Think about how these kinds of things can be achieved. What about input with a specific format such as dd-Mmm-yyyy?

[1] For example, Windows provides select-all by double-clicking the mouse and select-given-range, by clicking once and dragging.

[2] For example, Windows provides a pop-up menu when we right-click the mouse. Options include Cut, Copy, Paste, Delete, and Select All.

4.3 Text Areas

A *text area* is very much like a text field except that a text area can have more than one row and it can have a horizontal or vertical scrollbar, both, or neither. Like `TextField`, `TextArea` is a direct subclass of `TextComponent`.

The following display, produced by Cmta01.java, solicits input from the user in the text field at the bottom. When the Enter/Return key is pressed, the input string is appended to that in the text area along with a trailing newline. The text area is editable by default.

Here are the relevant parts of Cmta01.java:

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

class Cmta01 extends Frame implements ActionListener
{
/*1*/     private TextArea text   = new TextArea("First line.\n", 5, 20);
          private TextField input = new TextField(20);
```

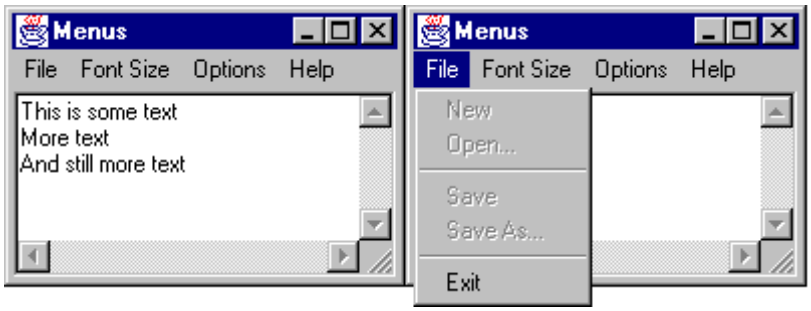
In case 1, we construct a text area, specifying its initial text contents and the number of rows and columns. As we learned in §4.2, the column count is only adhered to when the components are not stretched. Since this example uses a FlowLayout manager, which doesn't change its components' sizes when stretched, the text area has exactly the display size requested. As we can see, the display above shows 5 rows with up to 20 characters each. Since some lines are longer than 20 characters and more than 5 lines have been entered, both scrollbars are active. `TextArea.getColumns` behaves as described for text fields and `TextArea.getRows` behaves in a corresponding fashion.

5. Menus

In this chapter, we will see how to implement pull-down menus, dialogs and file dialogs, shortcut menus, and popup menus.

5.1 Introduction

Let us look at the displays below, produced by program Mn01.java:



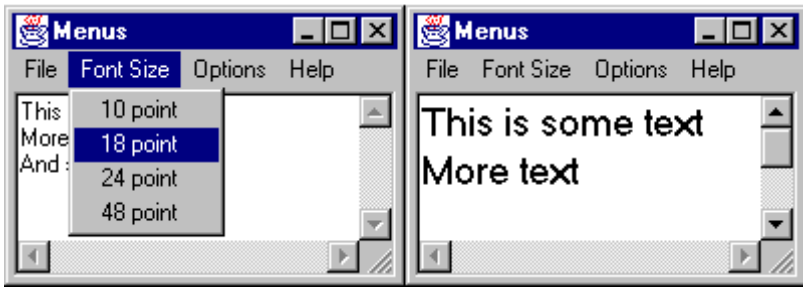
While buttons are convenient for triggering certain kinds of actions, they do take up screen space. For actions that are used less frequently, we might wish to provide a way to select those actions on demand, but have them remain out of sight the rest of the time. We can do this by implementing such actions items in a *menu*, which we pull down. For example, in the left frame above, four menus are displayed: File, Font Size, Options, and Help. These menus are part of a *menu bar* that has been attached to the parent frame. If the frame is not wide enough to have the whole menu bar on the same line, the menu bar is wrapped to the next line.

Apart from the menu bar, the only component associated with the frame is a text area, which is created with three lines of text set in 10-point plain type. The text area is read-only; that is, it has been made non-editable.

We can cause a menu to drop down by selecting it and, as we can see in the right frame above, the File menu contains five named entries each of which is referred to as a menu item}. Since the first four are grayed out we can see they have been disabled and, therefore, cannot be selected. Separators have been placed between the second and third items and also between the fourth and fifth. Like disabled items, separators are displayed in some alternate way and cannot be selected; separators serve only to visually separate groups of related items. Item Exit is enabled and choosing it results in the program's termination.

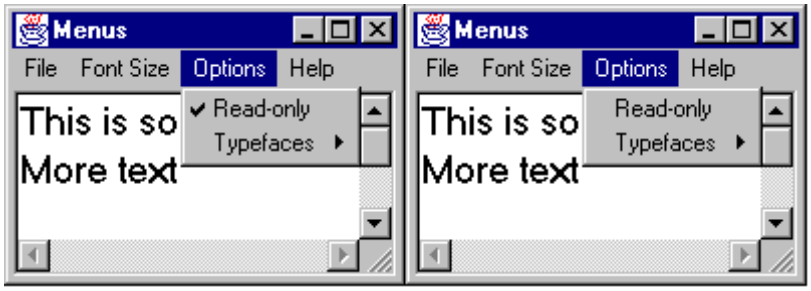
While the first four items are disabled (in fact, they are not even implemented), note that two of them have a ... suffix. On some systems this is used to indicate that choosing such an item results in more choices, in this case probably a file name.

The width of a drop-down menu is determined by the widest item label it contains.



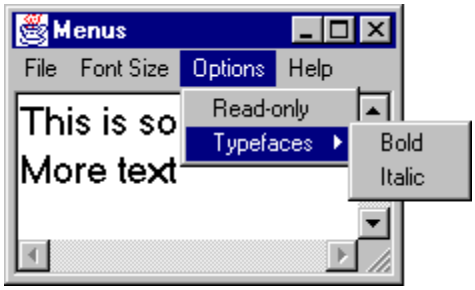
In the left frame above, we see the Font Size menu pulled down and the 18-point item about to be selected. When that item is selected, the event generated is detected and the text area's contents are set to that size type, as shown in the right frame.

An important property of the Font Size menu is that all of its items are mutually exclusive; the text cannot have more than one size simultaneously. However, some choices are binary and can be implemented a little differently; for example:

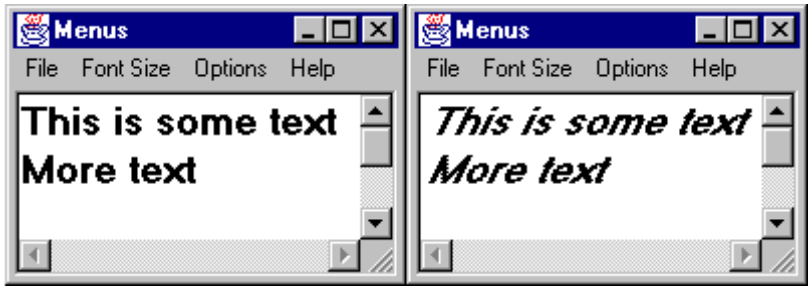


In the left frame we see the Options menu—we're only interested in the first item for now. When the program starts up, it puts the text area in read-only mode by disabling editing and then it places a checkmark next to this item. As such, this item is called a *checkbox menu item*. Each time we select this option its checked status toggles. For example, it was initially checked then when it was selected, it became unchecked as shown in the right frame above.

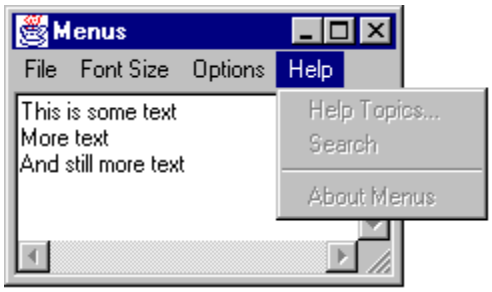
Look closely at the Typefaces item in the Options menu; it has a small black triangle to the right. This is used on Microsoft's systems to indicate that item is really a submenu. (Other systems may use a different indicator.) If we select this option, we see the following display:



As we can see, menus can nest and this one contains two items, both of which are checkbox menu items, since text can be either bold or italic, both, or neither. There is no limit on the depth of menu nesting, however, beyond two or three levels nested menus can become cumbersome to navigate. Here are some examples of selecting bold and bold/italic, respectively, both at 18 point:



The final display shows the Help menu. And as we can see, all of its items are disabled, since they have not yet been implemented. The menu also contains a separator. While this menu is just like any other, it is dealt with in a slightly special way to accommodate the fact that some windowing systems treat Help facilities specially. We'll discuss this later when we look at the program code.



5.2 The Example Source Code

Here then is the source code for Mn01.java:

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

class Mn01 extends Frame implements ActionListener, ItemListener
{
/*1*/ private MenuBar mb = new MenuBar();
/*2*/ private Menu menuFile = new Menu("File");
    private MenuItem menuFileNew = new MenuItem("New");
/*3*/ private MenuItem menuFileOpen = new MenuItem("Open...");
    private MenuItem menuFileSave = new MenuItem("Save");
    private MenuItem menuFileSaveAs = new MenuItem("Save As...");
    private MenuItem menuFileExit = new MenuItem("Exit");
    private Menu menuFont = new Menu("Font Size");
    private MenuItem menuFont10 = new MenuItem("10 point");
    private MenuItem menuFont18 = new MenuItem("18 point");
    private MenuItem menuFont24 = new MenuItem("24 point");
    private MenuItem menuFont48 = new MenuItem("48 point");
```