Java Programming: Part 2

ACKNOWLEDGMENTS

Content Development

The content of this self-study guide is based on the training courses "Java Programming" and "Advanced Java Programming," developed by Instruction Set, Inc. for its curriculum of instructor-led technical training. This guide was designed and developed by an Instruction Set team of instructional designers, course developers, and editors.

Administration

Vice President and General Manager of ZD University: Ed Passarella
Marketing Director: Risa Edelstein
Director, ZD University: George Kane
Senior Editor, Curriculum: Jennifer Golden
Project Director, Instruction Set: Laurie Poklop
Project Manager, Instruction Set: Sandy Tranfaglia

DISCLAIMER

While Ziff-Davis Education takes great care to ensure the accuracy and quality of these materials, all material is provided without any warranty whatsoever, including, but not limited to, the implied warranties of merchantability or fitness for a particular purpose.

Trademark Notices: ZD University and Ziff-Davis Education are trademarks and service marks of Ziff-Davis Inc. Java Programming: Part 2 is a Copyright of Instruction Set, Inc. All other product names and services used throughout this book are trademarks or registered trademarks of their respective companies. The product names and services are used throughout this book in editorial fashion only and for the benefit of such companies. No such use, or the use of any trade name, is intended to convey endorsement or other affiliation with the book.

Copyright © 1998 Instruction Set, Inc. All rights reserved. This publication, or any part thereof, may not be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, storage in an information retrieval system, or otherwise, without the prior written permission of Instruction Set, Inc., 16 Tech Circle, Natick, MA 01760, (508) 651-9085, (800) 874-6738. Instruction Set's World Wide Web site is located at http://www.InstructionSet.com. ZD University's World Wide Web site is located at http://www.zdu.com.

Photocopying any part of this book without the prior written consent of Instruction Set, Inc. is a violation of federal law. If you believe that Instruction Set materials are being photocopied without permission, please call 1-800-874-6738.
Java Programming: Part 2

Lesson 1: Events

Objectives .................................................................................. 2
Events ......................................................................................... 2
The Java Event Models ............................................................... 3
The Delegation Event Model—Action Events ......................... 4
  The ActionEvent Class ........................................................... 6
  Adjustment Events ................................................................. 7
  The AdjustmentEvent Class ................................................... 8
Item Events .................................................................................. 9
  The ItemEvent Class ............................................................... 11
Low-Level Event Listener Interfaces ....................................... 11
  The ComponentListener Interface ........................................ 13
  The ComponentEvent Class .................................................. 14
  Event Adapter Convenience Classes .................................... 14
  The FocusEvent Class ........................................................... 15
  The InputEvent Class ............................................................ 15
  The KeyEvent Class ............................................................... 16
  The MouseEvent Class .......................................................... 17
  The WindowEvent Class ....................................................... 17
Event Class Hierarchy .............................................................. 18
  The AWTEvent Class ............................................................ 18
The Extended Component Event Model ................................ 19
Lesson Summary ........................................................................ 21
Review Questions ....................................................................... 22
Exercise ..................................................................................... 23
LESSON 2: IMAGES, ANIMATION, AND AUDIO

Objectives ................................................. 26
Using Images ............................................. 26
  Locating Images ........................................ 26
  Loading Images .......................................... 27
  Displaying Images ....................................... 28
  Getting the Size of an Image .......................... 29
Animation and Multi-Threading ................................ 30
  Creating a Thread ....................................... 30
  Controlling a Thread .................................... 31
  Animating an Applet .................................... 32
  The start() and stop() Methods ....................... 34
Audio Clips .................................................. 35
Lesson Summary .......................................... 37
Review Questions ......................................... 38
Exercise ..................................................... 39

LESSON 3: ABSTRACT WINDOWING TOOLKIT BASICS

Objectives ................................................ 42
Introduction to AWT Components .......................... 42
  A Partial AWT Class Hierarchy ......................... 43
Labels ....................................................... 44
Buttons ..................................................... 46
  Action Events ............................................ 47
Text Components .......................................... 49
  Class TextField ......................................... 50
  Read-only TextFields .................................... 50
  Class TextArea ........................................... 51
Lists ......................................................... 53
  List Events .............................................. 54
Choice Menus ............................................... 55
  The PopupMenu Class .................................... 56
Checkboxes .................................................. 56
  CheckboxGroup ............................................ 58
Scrollbars .................................................. 58
  Scrollbar Events ........................................ 60
Layout ....................................................... 61
  Layout Managers ....................................... 62
Lesson 4: Application GUI

Objectives ................................................. 82
The Canvas Component ............................ 82
Frames ...................................................... 83
Class BorderLayout ................................... 84
Menus ...................................................... 86
Dialog Boxes .......................................... 88
Standalone Windows Applications .......... 89
Lesson Summary ................................. 91
Review Questions ............................. 92
Exercise ............................................. 93

Lesson 5: Exceptions

Objectives ................................................. 96
Traditional Error Handling ................. 96
Java Exceptions ...................................... 96
Creating an Exception Class ............... 97
The throw Statement ......................... 98
Defining an Exception Handler .......... 99
The catch Statement ......................... 100
The finally Statement .................... 101
Exception Types .......................... 102
The throws Clause ......................... 103
Lesson Summary ............................. 105
Review Questions .......................... 106
Exercise ............................................ 107
# Introduction ................................................. 156
# Java Beans Overview ........................................ 157
  Simple Java Bean ............................................ 159
  The BeanBox Tool .......................................... 160
# Events ......................................................... 161
  Events Object Interaction .................................. 162
  Event State Objects ...................................... 163
  Event Listener Interfaces .................................. 164
  Registering Event Listeners .............................. 165
  Unicast Event Sources ..................................... 166
  Firing the Event ........................................... 168
  Event Adaptors ............................................. 169
# The Core Reflection API ...................................... 171
# Properties .................................................. 172
# Indexed Properties ........................................ 173
  Bound Properties .......................................... 174
# Object Serialization ........................................ 176
  Serializable vs. Externalizable Interfaces ............. 177
  Serialization Examples .................................... 178
# JAR Files .................................................... 179
# Introspection ................................................ 180
  Introspection Design Patterns ......................... 180
  Implementing the BeanInfo Interface ................... 181
  BeanInfo Example ......................................... 183
# Lesson Summary ............................................. 184
# Review Questions .......................................... 185
# Exercise ....................................................... 186

## Appendix A: Hypertext Markup Language (HTML)

| HTML History and SGML ................................... 188 |
| Structure ................................................... 189 |
| Head Elements ............................................. 189 |
| Formatting: Blocks and Separators ...................... 190 |
| Formatting: Physical ..................................... 190 |
| List .......................................................... 191 |
| Netscape List Extensions ................................ 191 |
| Links .......................................................... 192 |
| Images ........................................................ 192 |
| Forms .......................................................... 193 |
| Tables (HTML 3) ............................................ 194 |
APPENDIX B: JAVA SAMPLE

Usage and Copyright Notification .......................... 198
The XYZApp.java Source .................................. 199

APPENDIX C: JAVA CLASS HIERARCHY

Java Class Hierarchy ....................................... 210

ANSWERS: TO REVIEW QUESTIONS

Lesson 1 ...................................................... 220
Lesson 2 ...................................................... 220
Lesson 3 ...................................................... 220
Lesson 4 ...................................................... 221
Lesson 5 ...................................................... 221
Lesson 6 ...................................................... 221
Lesson 7 ...................................................... 222
Lesson 8 ...................................................... 222
LESSON 1

Events

OVERVIEW
Events are the means by which a Java program interacts with the user. When the user performs an event, such as clicking the mouse, the application or applet that has been listening is directed to the appropriate process. There are many ways to interact with a Java program. As a result, there are many categories of events and many interfaces for each category. This lesson covers these interfaces and the means used to implement them.

LESSON TOPICS
- Events
- The Delegation Event Model—Action Events
- Item Events
- Low-Level Event Listener Interfaces
- Event Class Hierarchy
- The Extended Component Event Model
OBJECTIVES

By the end of this lesson, you should be able to:

- Enable an applet respond to input events.
- Understand the different event classes.
- Describe the event class hierarchy.

EVENTS

When the user interacts with an applet, the Java Abstract Windowing Toolkit (AWT) generates an event object and delivers it. Event objects are of class `java.awt.Event`. For example, when the user clicks or moves the mouse within the panel of an applet, the Java AWT generates an Event object that holds information about the mouse event, and looks for an event handler. Unless the applet contains one or more AWT components (discussed in a later lesson), the applet will receive every event generated. By adding event handling methods to the applet, a programmer can make an applet respond to events and behave interactively.

Events fall into several categories which are listed in Table A.

<table>
<thead>
<tr>
<th>Category</th>
<th>General Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>The user has performed an action such as a Button or MenuItem click</td>
</tr>
<tr>
<td>Mouse</td>
<td>The user has moved or clicked the mouse</td>
</tr>
<tr>
<td>Key</td>
<td>The user has pressed or released a key on the keyboard</td>
</tr>
<tr>
<td>Window</td>
<td>The containing window has been moved, closed, minimized or maximized</td>
</tr>
<tr>
<td>Focus</td>
<td>The user has shifted the input focus</td>
</tr>
<tr>
<td>Component-generated</td>
<td>An AWT component has generated an event, possibly after partially processing an input event (AWT components are discussed in a later lesson)</td>
</tr>
<tr>
<td>User-defined</td>
<td>Everything else</td>
</tr>
</tbody>
</table>

The `Event` class defines an integer field, named “id”, that identifies the specific type of event. Standard event types are defined within the `Event`
class as public constants with names such as `KEY_PRESSED` and `MOUSE_MOVED`. The default event handler of class `Applet` is the `processEvent()` method, inherited from class `java.awt.Component`. The default handler checks the `id` field of the event and dispatches events of certain standard types to one of several secondary handler methods. For example, if the event `id` is `MouseEvent.MOUSE_MOVED`, then the default handler calls a `processMouseMotionEvent()` method, passing the original `MouseEvent` object. To handle events, an applet may override `processEvent()`, override one or more secondary handler methods, or override both. An applet must also enable the events by calling `enableEvents()` or one of the methods to add an event listener.

The Java Event Models

Java Development Kit (JDK) 1.1 introduced two new models associated with event handling:

- Delegation Event Model
- Extended Component Event Model

A new set of classes, called listeners, was also introduced whose purpose is to listen for certain event types to occur. These classes define a method (for example, `actionPerformed()`) which is used to define the activity that is to take place when a particular action occurs.

The listener classes listen for two types of events: low-level and semantic.

Low-level events are the traditional events (e.g., mouse events) that happen to a GUI element in a windowing system. Listeners may handle low-level events before the component that originated them does, allowing a listener to consume the event so that it never gets handled by the originating component.

Semantic events are higher-level events that a user can perform, such as pressing the `<ENTER>` key or selecting a menu item. They usually indicate that a component has changed its value.
The semantic event listeners and their associated methods are listed in Table B.

**Table B. Semantic Event Listeners and their Associated Methods**

<table>
<thead>
<tr>
<th>Listener</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActionListener</td>
<td>actionPerformed (ActionEvent)</td>
</tr>
<tr>
<td>AdjustmentListener</td>
<td>adjustmentValueChanged (AdjustmentEvent)</td>
</tr>
<tr>
<td>ItemListener</td>
<td>itemStateChanged (ItemEvent)</td>
</tr>
</tbody>
</table>

**THE DELEGATION EVENT MODEL—ACTION EVENTS**

Events such as the clicking of a button or the movement of a scroll bar can be intercepted and processed by an applet. The technique used by Java to do this is called the Delegation Event Model. This model uses the concept that an object, such as a command button, is the source of events and another object, such as an applet, that listens to a source for a specific type of event.

A listener will inform the source of events that it is interested in listening to that source. Once informed, the source will add that listener to its list of objects that listens to the source. When an event occurs, the source will inform all interested listeners that the event has occurred and will pass to the listener information regarding the event.

If a listener is interested in an event it must implement a set of methods that the source can call when the event occurs. This set of methods is called an interface. The concept of an interface will be discussed in greater depth later in the course. Having implemented the interface, the source is then able to call the listener. When a method of the interface is called a single argument, an event object, is passed that conveyed information about the event.

The essential steps required for handling events include:

1. Using the `implements` keyword by the listener.
2. Adding the listener to the source’s event list.
3. Implementing the appropriate methods.
4. Using the event object to process the event.
The `implements` keyword is used in the class definition to inform the Java compiler that this class supports a specific type of interface. There are several types of event interfaces supported by Java. The `ActionListener` and `AdjustmentListener` interfaces are described in this section. The `ActionListener` interface deals with high level events such as the pressing of a button. The `AdjustmentListener` is used with such controls such as a scroll bar. Other high and low level interfaces exist.

Step 1: Implement the listener:

```java
public class MyApplet extends Applet implements ActionListener {
  public void init ()
  {
    Button b = new Button ("Help");
    b.addActionListener(this);
    add (b);
  }
}
```

Step 2: Add the listener to the source's event list:

```java
public void actionPerformed(ActionEvent evt)
{
}
```

The `addActionListener` method is used to add the source, the current applet, to the list of listeners interested in listening to the button. The `this` keyword identifies the current applet as the listener. The method is applied to the button thus making the button the source object.

Step 3: Implement the appropriate method:

```java
public void actionPerformed(ActionEvent evt)
{
}
```

The `ActionListener` interface has only a single method, `actionPerformed`. When an interface is implemented by a class, all of its methods must be overridden by that class. The `actionPerformed` method is passed a single argument, `ActionEvent`. This argument contains information about that event.
Step 4: Process the event:

```java
if (evt.getSource() instanceof Button) {
    Button source = (Button)evt.getSource();
    if (source.getLabel() == "Help")
        /* ... handle “Help” button ... */
}
```

**Warning**

In this example, if you don’t provide an implementation for the `actionPerformed` method, you will get a compiler error.

The `implements` keyword is used in the class definition to inform the Java compiler that this class supports a specific type of interface. There are several types of event interfaces supported by Java. The `ActionListener` and `AdjustmentListener` interfaces are described in this section. The `ActionListener` interface deals with high level events such as the pressing of a button. The `AdjustmentListener` is used with such controls such as a scroll bar. Other high and low level interfaces exist.

### The `ActionEvent` Class

The `ActionEvent` class contains the event type `ACTION_PERFORMED`. An `ActionEvent` can be fired by:

- clicking a button
- double-clicking a list
- choosing a menu item
- pressing `<ENTER>` when in a text field

The `getSource()` method is used to determine the object in which the event originated.

An action command is associated with each `ActionEvent`. This command is a simple String that is assigned to a source object. It is assigned when the object is created using the `setActionCommand` method. Later the `getActionCommand` method can be used within such methods as `actionPerformed` to uniquely identify the source of the event. This technique avoids having to use the label or some other attribute of the source that might change with the use of the object.

The user may be holding down a modifier key such as `<ALT>` or `<CTRL>` when an event is fired. The `getModifiers()` method can be used to...
determine which key was held down. The following masks are provided with the `ActionEvent` class for comparison with the modifier constants:

- `ALT_MASK`
- `CTRL_MASK`
- `META_MASK`
- `SHIFT_MASK`

**Adjustment Events**

Adjustment events are fired when the user manipulates an adjustable item, such as a scrollbar or slider control. They are handled similarly to action events.
For example, to respond to a scrollbar manipulation, implement the `adjustmentValueChanged` method of the `AdjustmentListener` interface, and query the `AdjustmentEvent` object:

```java
... import java.awt.event.*;
public class MyApplet extends Applet implements AdjustmentListener
{
    public void init ()
    {
        Scrollbar sb = new Scrollbar (...);
        sb.addAdjustmentListener(this);
        add (sb);
    }

    public void adjustmentValueChanged (AdjustmentEvent evt)
    {
        switch (evt.getAdjustmentType ())
        {
            case AdjustmentEvent.UNIT_INCREMENT:
            case AdjustmentEvent.UNIT_DECREMENT:
            case AdjustmentEvent.BLOCK_INCREMENT:
            case AdjustmentEvent.BLOCK_DECREMENT:
            case AdjustmentEvent.TRACK:
                Scrollbar sb = (Scrollbar)evt.getAdjustable();
                int orientation = sb.getOrientation ();
                int value = sb.getValue ();
                /* ...update display to match value */
        }
    }
}
```

### The AdjustmentEvent Class

The `AdjustmentEvent` class contains the event type `ADJUSTMENT_VALUE_CHANGED`. This event can be fired by:

- manipulating a scrollbar
- manipulating a custom component (such as a slider)

The `getAdjustable()` method returns a reference to the component that fired the event. If that type of adjustment is needed, the `getAdjustable()`...
ableType() method will return one of the following values which are defined in the AdjustableEvent class:

- BLOCK_DECREMENT
- BLOCK_INCREMENT
- TRACK
- UNIT_DECREMENT
- UNIT_INCREMENT

ITEM EVENTS

Item events are fired when the user selects or deselects an item in a List, Choice, Checkbox, CheckboxMenuItem, or CustomComponent.
To respond to an item event, implement the `itemStateChanged` method of the `ItemListener` interface, and query the `ItemEvent` object:

```java
import java.awt.event.*;
public class ListApplet extends Applet implements ItemListener
{
    public void init ()
    {
        List myList = new List (6, false);
        for (int i = 0; i < names.length; ++i)
        {
            myList.addItem (names[i]);
        }
        myList.addItemListener (this);
        add (myList);
    }
    public void itemStateChanged (ItemEvent evt)
    {
        if (evt.getStateChange() == ItemEvent.SELECTED)
        {
            List list = (List)evt.getSource();
            String item = list.getSelectedItem();
            /* ... work with the item ... */
        }
    }
}
```

The type of event is determined by `getStateChange()`. `SELECTED` and `DESELECTED` are possible return values.
The ItemEvent Class

The ItemEvent class contains the event type ITEM_STATE_CHANGED. An ItemEvent is fired by selecting or deselecting one of the following items:

- list
- choice
- checkbox
- checkbox menu item
- custom component

The getStateChange() method is used to determine the type of state change. This method returns one of the following values:

- SELECTED
- DESELECTED

The class also contains a getItemSelectable() method which returns a reference to the component that fired the ItemEvent. The getItem() method returns an object that identifies the item that was changed.

LOW-LEVEL EVENT LISTENER INTERFACES

The low-level listener interfaces allow the listener to respond to the low-level events of a windowing system. Low-level events are events, such as mouse events, which may be consumed by the listener before the source component receives them.
The low-level event listeners and their associated methods are:

**Component Listener**
- componentHidden (ComponentEvent)
- componentMoved (ComponentEvent)
- componentResized (ComponentEvent)
- componentShown (ComponentEvent)

**Focus Listener**
- focusGained (FocusEvent)
- focusLost (FocusEvent)

**Key Listener**
- keyPressed (KeyEvent)
- keyReleased (KeyEvent)
- keyTyped (KeyEvent)

**MouseListener**
- mouseClicked (MouseEvent)
- mouseEntered (MouseEvent)
- mouseExited (MouseEvent)
- mousePressed (MouseEvent)
- mouseReleased (MouseEvent)

**MouseMove Listener**
- mouseDragged (MouseEvent)
- mouseMoved (MouseEvent)
Window Listener

- windowClosed (WindowEvent)
- windowClosing (WindowEvent)
- windowDeiconified (WindowEvent)
- windowIconified (WindowEvent)
- windowOpened (WindowEvent)

The ComponentListener Interface

When a Component is moved, resized, shown, or hidden, an event will be sent to any ComponentListener added to that Component. The ComponentListener interface defines a method for each of the four Component event types:

```java
class MyComponentListener implements ComponentListener {
    public void componentHidden (ComponentEvent evt) {
        /* ... */
    }
    public void componentMoved (ComponentEvent evt) {
        /* ... */
    }
    public void componentResized (ComponentEvent evt) {
        /* ... */
    }
    public void componentShown (ComponentEvent evt) {
        /* ... */
    }
}
```

The low-level listeners have more methods than the semantic interfaces and normally require more work to implement. To avoid this, each low-level listener interface has an associated adapter class that implements an empty
stub version of every method in the interface. Extending the adapter class allows the listener class to implement only the desired method(s):

```java
class MyComponentListener extends ComponentAdapter
    implements ComponentListener
{
    public void componentHidden (ComponentEvent evt)
    {
        /* This is the only interesting component event */
    }
    /* Inherit other (stub) methods from ComponentAdapter */
}
```

### The `ComponentEvent` Class

The `ComponentEvent` class is the root class for all low-level event classes. It contains a method called `getComponent()` which can be used to identify the component that fired the event.

The `getId()` method can be used to determine which event type was fired. It is useful when using the Extended Component Event Model which will be discussed later.

The `ComponentEvent` event types are:

- `COMPONENT_HIDDEN`
- `COMPONENT_MOVED`
- `COMPONENT_RESIZED`
- `COMPONENT_SHOWN`

Any component in the Component Hierarchy can fire a `ComponentEvent`.

### Event Adapter Convenience Classes

Implementing an interface in a class means that each method defined by the interface must be defined in the class. The six low-level event listener interfaces all have more than one method, placing a burden on the programmer.
To assist with the problem, six classes corresponding to the six low-level event listener interfaces have been provided. Each class implements the corresponding event listener interface by defining empty versions of each required method.

The event adapter convenience classes are:

- ComponentAdaptor
- FocusAdapter
- KeyAdapter
- MouseAdapter
- MouseMotionAdapter
- WindowAdapter

**The FocusEvent Class**

The FocusEvent class contains the following event types:

- FOCUS_GAINED
- FOCUS_LOST

The getId() method can be used to determine which event type was fired.

**The InputEvent Class**

This is the root class for all input events. It contains methods which can be used to determine which modifier keys or mouse buttons were pressed at the time of the event.

The following methods return a boolean indicating whether a particular modifier key was pressed:

- isAltDown()
- isControlDown()
- isMetaDown()
- isShiftDown()
The `getModifiers()` method returns an integer which can be compared to the following masks to see which modifier keys or mouse buttons were pressed.

- ALT_MASK
- CTRL_MASK
- META_MASK
- SHIFT_MASK
- BUTTON1_MASK
- BUTTON2_MASK
- BUTTON3_MASK

The class also contains a `getWhen()` method which returns a timestamp of when the event occurred.

**The KeyEvent Class**

The `KeyEvent` class contains the following event types:

- KEY_PRESSED
- KEY_RELEASED
- KEY_TYPED

The `getId()` method can be used to determine which of these event types was fired.

The `getKeyCode()` method can be used to find out which key was pressed to fire the event. The method returns the key code of the appropriate key in integer form. If the key is an action key, the return code will match one of the action key constants defined in the `KeyEvent` class. The `isActionKey()` method can be used to determine if a key is an action key.

The `getKeyChar()` method can also be used. It returns the character associated with the key that was pressed when the event occurred.
The `MouseEvent` Class

The `MouseEvent` class contains the following event types:

- `MOUSE_CLICKED`
- `MOUSE_ENTERED`
- `MOUSE_EXITED`
- `MOUSE_PRESSED`
- `MOUSERELEASED`
- `MOUSE_DRAGGED`
- `MOUSE_MOVED`

The `getId()` method can be used to determine which of these events occurred.

The mouse position can be determined by calling either the `getPoint()` method or the `getX()` and `getY()` methods.

The `getClickCount()` method returns the number of mouse clicks associated with the event.

The `WindowEvent` Class

The `WindowEvent` class contains the following event types:

- `WINDOW_CLOSED`
- `WINDOW_CLOSING`
- `WINDOW_DEICONIFIED`
- `WINDOW_ICONIFIED`
- `WINDOW_OPENED`

The inherited `getId()` method is used to determine which of the events occurred.

The class contains a `getWindow()` method that returns a reference to the window that received the event.
EVENT CLASS HIERARCHY

The various event types are organized into a hierarchy of classes. These classes are contained in the `java.awt.event` package, even though they are all derived directly or indirectly from the `java.awt.AWTEvent` class.

```
EventObject
    AWTEvent
        ActionEvent
        AdjustmentEvent
        ItemEvent
        ComponentEvent
            FocusEvent
            InputEvent
                KeyEvent
                MouseEvent
            PaintEvent
            WindowEvent
```

The AWTEvent Class

The `AWTEvent` class is the top-level class for all AWT-related classes. It contains a method called `getId()` which enables the programmer to determine which event type was fired.

The `AWTEvent` class also contains some masks which are used with the Extended Component Event Model (JDK 1.1). These masks are used to
determine which events should be delivered to a component's event processing methods. The masks include:

- ACTION_EVENT_MASK
- ADJUSTMENT_EVENT_MASK
- ITEM_EVENT_MASK
- COMPONENT_EVENT_MASK
- FOCUS_EVENT_MASK
- KEY_EVENT_MASK
- MOUSE_EVENT_MASK
- MOUSE_MOTION_EVENT_MASK
- WINDOW_EVENT_MASK

### The Extended Component Event Model

The Extended Component Event Model offers a way of handling events that originate in AWT components. This model uses the default event handling methods in the Delegation Event Model. Those methods and the `processEvent()` method can be used by overriding them in the components that are extended from AWT components.

The `enableEvents()` method from the AWT Component class allows events to be passed to the `processEvent()` method.
If the component’s `enableEvents()` method is called with one or more of the previously defined masks, the associated event types will be passed to the `processEvent()` method.

The following code allows an applet to receive all mouse events:

```java
public void init ()
{
    enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                 AWTEvent.MOUSE_MOTION_EVENT_MASK);
}
public void processEvent (AWTEvent evt)
{
    switch (evt.getID())
    {
    case MouseEvent.MOUSE_ENTERED:
       /* ... */
       break;
    case MouseEvent.MOUSE_EXITED:
       /* ... */
       break;
       /* ... */
    }
```
LESSON SUMMARY

In this lesson, you have learned:

- Events are used to interact with the user.
- There are several categories of events in Java:
  - Action
  - Mouse
  - Key
  - Window
  - Focus
  - Component-generated
  - User-defined
- The Semantic Event Listener Interface handles high level events such as Action Events, Adjustment Events, and Item Events.
- Traditional events, such as mouse events, are handled by the Low-Level Event Listener Interface.
1. What are the methods associated with each semantic event listener?

2. What method determines the origin of an action event?

3. What is the purpose of the Extended Component Event Model?

Answers on page 220
**EXERCISE**

1. Write an applet that contains a draggable rectangle. The applet’s parameters are `rectwidth`, `rectheight` and `rectcolor`, specifying the width, height and color of the rectangle. The user may click within the rectangle and drag it to a new position. The applet should handle `MOUSE_PRESSED`, `MOUSE_DRAGGED`, and `MOUSE_RELEASED` events to implement this. Either call `enableEvents`, or implement the `MouseListener` and `MouseMotionListener` interfaces.
LESSON 2

Images, Animation, and Audio

OVERVIEW
The Java language includes special methods for using images and sound in programs. A graphic or sound file is located using a URL string as if looking up a Web address. The file is then loaded into an Image or AudioClip object and handled accordingly. In order to handle animation, it is often necessary to use multi-threading. Since each thread runs independently of other threads, a program can juggle many processes simultaneously.

LESSON TOPICS
- Using Images
- Animation and Multi-Threading
- Audio Clips
OBJECTIVES

By the end of this lesson, you should be able to:

- Use graphics library methods to draw in an applet.
- Understand and implement threads.
- Include audio clips in an applet.

USING IMAGES

The Java AWT includes facilities for bitmapped graphics, including a framework for loading images from files of various formats.

Currently, image loaders for GIF and JPEG formats are provided with Java. Java may provide support for more formats in the future.

Image use is discussed in three steps:

1. locating image files on the network
2. loading image files
3. displaying images

Locating Images

The method Applet.getImage() returns an image object giving the network location of the file that contains the image. The network location is an object of class URL. A URL object represents a Uniform Reference Locator string, the common means of referring to an object on the World Wide Web.

A URL object can be constructed from a string, as in:

```java
URL location = new URL("http://not.a.real.site/java");
```

Note that class URL resides in package java.net, so if variables of type URL are used, the class must be imported.
Since the image files used by an applet usually reside at a location relative to the location of the applet's code, class Applet provides methods that allow URL objects to represent these locations.

- `getDocumentBase()` returns the URL of the document's directory.
- `getCodeBase()` returns the URL of the applet's code base.

By using one of these methods to locate images, the applet can move to a new location without changing the code.

### Loading Images

Java image objects are of class `java.awt.Image`.

Here is a call to `Applet.getImage()`:

```java
Image myImage = getImage (getCodeBase(), "images/myface.gif");
```

Note that this form of `getImage()` takes its URL in two pieces; the first argument, of type `URL`, is the base location; the second argument, of type `String`, is a relative path. This form is more convenient than the single-argument form if the base URL is a call to `getDocumentBase()` or `getCodeBase()`.

The `getImage()` method returns immediately. The `Image` object it returns contains the information necessary for loading the image, but does not yet refer to the image data. If the image location is bad, `getImage()` does not report an error. The Java AWT library loads image data on demand. When the AWT is first asked to display an image or extract information from an image, the AWT commences the load. At that time, the applet may check for errors.

Since the AWT loads image data on demand, it is appropriate to get all the image objects an applet needs ahead of time, using the `init()` method. Doing so will not delay the applet's startup process.
Displaying Images

To display an image, insert a call to one of the `Graphics.drawImage()` methods in an applet's `paint()` method. For example:

```java
public class ImageApplet extends Applet {
    Image myImage;
    int imageX, imageY;
    /* ... */
    public void paint (Graphics g) {
        g.drawImage (myImage, imageX, imageY, this);
    }
}
```

This form of `drawImage()` draws the image positioned with its upper left corner at the given `(x, y)` position. This form does not scale the image.

In the example above, the fourth parameter to `drawImage()` is an image observer. The type of an image observer is interface `java.awt.image.ImageObserver`. When an applet loads image data by calling `Graphics.drawImage()`, `Image.getWidth()`, or any one of a number of other methods, the image data becomes available one piece at a time. As each piece becomes available, AWT reports the results through the `ImageObserver` interface. The image observer the caller provides may be null, but if the observer is an object, that object receives notification of the image load and can respond to it. For instance, when more image pixels become available, the image observer can issue a call to `repaint()` to display the new pixels.

In many cases, the image observer is the applet itself. Fortunately, all AWT components implement the `ImageObserver` interface. The default implementation of `ImageObserver` provided by `java.awt.Component` is suitable for most purposes.

A second form of `Graphics.drawImage()` allows the caller to scale the image to a specific width and height:

```java
int x, y, width, height;
g.drawImage (myImage, x, y, width, height, this);
```
Getting the Size of an Image

Like the pixel data of an image, the image's width and height is not immediately available from an image handle. Class Image provides getWidth() and getHeight() methods, but these do not necessarily return the width and height of the image. Instead, they take an ImageObserver argument and report the size information back to the ImageObserver object when that information becomes available.

Here is the skeleton of an applet that gets the height and width of an image:

```java
import java.applet.*;
import java.awt.*;
import java.awt.image.ImageObserver;
public class CatAndMouse extends Applet
implement ImageObserver
{
    Image myImage;
    Dimension imageSize = new Dimension (0, 0);
    /* ... */
    public void init ()
    {
        /* ... */
        // Calls to Image.getWidth() and Image.getHeight():
        imageSize.width = myImage.getWidth (this);
        imageSize.height = myImage.getHeight (this);
    }
    // Pick up the image width and/or height when either
    // becomes available.
    //
    public boolean imageUpdate (Image img, int flags,
    int x, int y, int width, int height)
    {
        if ((flags & WIDTH) != 0)
            imageSize.width = width;
        if ((flags & HEIGHT) != 0)
            imageSize.height = height;
        return super.imageUpdate (img, flags, x, y,
            width, height);
    }
}
```

ImageObserver contains only one method:

```java
public boolean imageUpdate (Image img,
    int infoflags, int x, int y,
    int width, int height);
```
In addition, interface ImageObserver defines these flags:

```java
public static final int WIDTH = 1;
public static final int HEIGHT = 2;
```

When the width of the image is available, there is a call to `imageUpdate()` and the flag WIDTH is set.

---

**ANIMATION AND MULTI-THREADING**

The next topic is animation; however, since Java threads are essential to animation, the basics of threads are discussed first.

Java multi-threading allows several execution threads to run simultaneously in parallel. Each thread is a separate process, but all threads within a single Java program share the same data space. Threads are important to animation because some flow of control must be active to drive an animated sequence frame by frame.

Java multi-threading can be used to create animated displays that run independently of window event handling and other tasks.

The example used in this lesson is `CatAndMouse`, an applet in which an image follows the movement of the cursor.

**Creating a Thread**

A Java thread is represented by an object of type `java.lang.Thread`:

```java
Thread myThread;
```

To start a new thread, a `Thread` object is constructed and its `start()` method is called. The thread begins by executing a method called `run()`. This method is either the `run()` method of the thread itself, or the `run()` method of another object, depending on how the thread was constructed. The other object must implement the `java.lang.Runnable` interface:

```java
Runnable runMe = /* ... */;
Thread myThread = new Thread (runMe);
```
In most cases, the animation thread of an applet executes the `run()` method of the applet itself. In this case, the applet implements `Runnable`:

```java
class Loader implements Runnable {
    public void run () {
        /* Code to run goes here */
    }
}
```

The implementation of the `run()` method depends on the application. In an applet that contains animation, the `run()` method is usually a loop in which the thread sets up a display frame, calls `repaint()`, and then goes to sleep for a short time before iterating.

## Controlling a Thread

The following sample shows the signatures of the `start()`, `stop()`, and `sleep()` methods:

```java
public class Thread {
    public void start() {} 
    public void stop() {} 
    public static void sleep(long millis) {
        throws InterruptedException {} 
        public static void sleep(long millis, int nanos) {
            throws InterruptedException {} 
    }
}
```

A thread is started by calling its `start()` method:

```java
Thread myThread;
/* ... */
myThread.start (); // returns immediately
```

A thread is killed by calling its `stop()` method:

```java
myThread.stop ();
```
A thread can sleep for a specified number of milliseconds by calling Thread.sleep():

```java
try {
    Thread.sleep (1000); // one second
} catch (InterruptedException ie) {
}
```

**Animating an Applet**

The CatAndMouse example is a simple animation applet. In it, an image appears to follow the cursor. The following excerpt of
CatAndMouse.java illustrates the main points of setting up an animation thread:

```java
public class CatAndMouse extends Applet
    implements Runnable, ImageObserver {
    /* ... */
    Thread animator;
    public void init ()
    {
        /* ... */
        // Construct a thread for animation
        animator = new Thread (this);
    }
    public void start ()
    {
        animator.start ();
    }
    public void stop ()
    {
        animator.stop ();
    }
    public void run ()
    {
        // Wake up every tenth of a second to move the image.
        //
        for (;;) {
            if (updateImageLocation ()) {
                repaint ();
            }
            try {
                Thread.sleep (100);
            } catch (InterruptedException ie) {
                break;
            }
        }
    }
    private boolean updateImageLocation () { ... }
}
```

The applet's `run()` method accomplishes the following in a loop:

- updates the display parameters
- calls `repaint()`
- sleeps for a period
The loop from `CatAndMouse` appears below:

```java
public void run ()
{
    for (;;) {
        if (updateImageLocation ()) {
            repaint ();
        }
        try {
            Thread.sleep (100);
        } catch (InterruptedException ie) {
            break;
        }
    }
}
private boolean updateImageLocation () { ... }
```

### The `start()` and `stop()` Methods

Class `Applet` provides a `start()` method. This is not to be confused with the `start()` method of class `Thread`. The prototype is as follows:

```java
public void start () {}
```

The viewer or browser, after calling `init()`, calls `start()` to start up any threads the applet needs to run. In the `CatAndMouse` example, the `start()` method does indeed start up a thread. The thread was constructed previously in the `init()` method.

Class `Applet` provides a corresponding `stop()` method that the viewer or browser calls when the window is minimized, when the applet is no longer on the current page, or in any other circumstance in which it is appropriate to stop the applet’s threads. Its prototype is shown below:

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

If the user brings up the applet again, and the applet is still loaded, the viewer or browser can call `start()` again to restart the applet, rather than reloading it.
To finish animating `CatAndMouse`:

```java
public void start ()
{
    animator.start ();
}

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

### AUDIO CLIPS

The applet package also supports audio playback, allowing the integration of sound into applets. The procedure for working with audio is similar to that for locating and loading images.

The `Applet.getAudioClip()` method returns a `java.applet.AudioClip` object when given a URL. It is similar in function to `Applet.getImage()`:

```java
AudioClip noise =
    getAudioClip (getCodeBase(), “audio/noise.au”);
```

Once an `AudioClip` object is constructed, the methods listed in Table C can be invoked on it.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>play()</td>
<td>Start playing this clip from the beginning. Stop at the end.</td>
</tr>
<tr>
<td>loop()</td>
<td>Start playing this clip in a repeating loop.</td>
</tr>
<tr>
<td>stop()</td>
<td>Stop looping this clip.</td>
</tr>
</tbody>
</table>
An audio clip can be played in response to an event:

```java
public boolean keyDown (Event evt, int key) {
    if (key == '\n')
        noise.play ();
    return true;
}
```

A clip can be played in a continuous loop:

```java
noise.loop ();
```

Stop the loop deliberately by using the following:

```java
noise.stop ();
```

The danger with looping an AudioClip is that the loop continues until a call to stop() is made. Unless care is taken to stop all loops in progress when an applet stops, the browser continues to play them indefinitely. If an applet plays one or more audio clips in a loop, all of them must be halted in the applet's stop() method.
LESSON SUMMARY

In this lesson, you have learned:

- Graphics in Java are located using URL objects, loaded using the Image type, and displayed using drawImage() methods.

- The method ImageObserver is used to get the size of an image. It reports the width and height as soon as these values are available.

- Java multi-threading enables a flow of execution to run independently of the rest of the program.

- The AudioClip class facilitates the inclusion of sound into an applet.
1. What does the applet below do?

```java
public MysteryApplet extends Applet
    implements Runnable
{
    public void start () {
        Thread t = new Thread (this);
        t.start ();
    }
    public void run () {
    try {
        Color c = Color.red;
        for (;;) {
            Thread.wait (1000);
            setBackground (c);
            c = c == Color.red ? Color.blue : Color.red;
        }
    }
    catch (InterruptedException ie) {}
    }
}
```

2. What is wrong with the following code?

```java
String location = new String("http://someurl.com/image.gif");
Image graphic = getImage(location);
```

3. What is the danger with looping an AudioClip?

Answers on page 220
EXERCISE

1. Write an applet to present a slide show that cycles through a set of images, displaying one after another. The applet takes a parameter named `duration` that specifies the number of seconds to display each image. The default value of `duration` is 4. The applet also takes a number of parameters named `image0`, `image1`, `image2`, ... specifying the image files to load. The applet begins with `image0` and loops, adding one to the number in the parameter name each time, until the parameter value is null. The applet scales each image to fill the panel. (Hint: create a thread.)
LESSON 3

Abstract Windowing Toolkit Basics

OVERVIEW

As seen in the previous lessons, the Java Abstract Windowing Toolkit (AWT) includes facilities for rendering text and graphics, handling input events, and loading images. The AWT also includes facilities for building graphical user interfaces from a set of standard component classes. This lesson covers the component classes of the AWT.

LESSON TOPICS

- Introduction to AWT Components
- Labels
- Buttons
- Text Components
- Lists
- Choice Menus
- Checkboxes
- Scrollbars
- Layout
OBJECTIVES

By the end of this lesson, you should be able to:

➤ Construct AWT components such as buttons, lists and menus.
➤ Handle user interaction with AWT components.
➤ Adjust the layout of AWT components.
➤ Understand the concept of absolute layout.

INTRODUCTION TO AWT COMPONENTS

The design goals of the Abstract Windowing Toolkit (AWT) are in line with the goals of the Java language. The AWT is designed to be:

➤ simple
➤ platform-neutral

In general, the AWT allows the native windowing system to dictate the exact appearance and behavior of each Graphical User Interface (GUI) component while providing a programming interface to those features of the component that are common across all platforms. In other words, the AWT allows the programmer to make use of the “least common denominator” across popular windowing systems. Many features available on a particular platform, such as Windows NT, but not across all Java platforms, are not available through the AWT. Standard AWT components include the vari-

TRIVIA

While this least common denominator approach may sound like a disadvantage, this is the price you pay for portability. There is no such thing as a free lunch!
ous controls and views that traditionally make up a graphical user interface.
These include:

- labels
- buttons
- text boxes
- list boxes
- pull-down menus
- check boxes
- radio boxes
- scroll bars

Windows themselves are also AWT components. In general, a window or panel that can accommodate other components is called a container. `java.awt.Container`, derived from `java.awt.Component`, is the base class for all AWT container types. AWT container types include:

- dialogs
- frames
- panels

The basic use of each of these components is covered in this lesson. Understanding how to use a component means understanding:

- how to construct an object of the component type
- what methods are available on the component
- what events the component generates

**A Partial AWT Class Hierarchy**

Please refer to Appendix C, Java Class Hierarchy, for a more complete representation of the AWT Class Hierarchy. A partial hierarchy is shown in the Figure 1.
FIGURE 1  
The AWT class hierarchy

LABELS

A label is an object that represents a drawable text string. Its type is java.awt.Label, which is derived from java.awt.Component. Because it is a self-contained object, a label contains all the information necessary to draw itself, including its text, font, size and position. Like all AWT components, once added to a container, a label assumes responsibility for drawing itself. Working with a Label object can be less complicated than drawing text through graphics calls.
Class `Label` has three constructors:

- `public Label ()`
- `public Label (String text)`
- `public Label (String text, int alignment)`

The alignment of a `Label` can be one of the following:

- `Label.LEFT`
- `Label.CENTER`
- `Label.RIGHT`

Class `Label` also provides the following methods:

- `public int getAlignment ()`
- `public void setAlignment (int alignment)`
- `public String getText ()`
- `public void setText (String text)`
- `public Font getFont ()`
- `public void setFont (Font font)`

The following example sets a label's text, font, and color:

```java
lab.setText ("Your ad here");
lab.setFont (new Font ("Courier", Font.BOLD, 9));
lab.setForeground (Color.blue);
```
Here is the “Hello, World” applet, used in Java Programming: Part 1, rewritten to use a `Label`. The `add()` method of the applet adds a component (in this case a `Label`) to the applet:

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

public class LabelApplet extends Applet {
    public void init () {
        Label l = new Label ("Hello, World");
        l.setFont (new Font ("Helvetica", Font.BOLD, 18));
        add (l);
    }
}
```

### Buttons

Class `java.awt.Button` represents a pushbutton control. The class has two constructors:

- `public Button ()`
- `public Button (String label)`

A button is added to an applet in the same way as a `Label`:

```java
public void init () {
    Button b = new Button ("Cancel");
    add (b);
}
```

The button appears to depress when it is clicked. That much event processing happens automatically within the `Button` class. However, for a button to be useful, one must be able to attach code to the pressing of the button. This is accomplished by handling an action event.
Action Events

Within the Button class, event handling code handles mouse events, watching for a (mouse-down, mouse-up) event sequence within the boundaries of the button. When this sequence occurs, the event handling code issues a new event to the Button. The id of the Event object is Event.ACTION_EVENT.

The basic Button class does nothing to handle action events. According to Java event processing logic, the event is handled either by an EventListener or by a class derived from Button. Typically, the container handles action events from its component.

The actionPerformed() method of the ActionListener interface may respond to button clicks:

```java
public void actionPerformed(ActionEvent evt)
```

The argument to actionPerformed() is the ActionEvent. When the action originates from a button, the ActionEvent object's getActionCommand() method returns the label of the button in the form of a String. The label is useful for determining which component generated the event. The source of the event can also be determined using getSource() method, which returns an object that refers to the button. (Of course, if the listener is added to only one possible source of action events, identifying the source of the event is trivial.)

Code may be attached to an action by overriding the action() method in the applet class:

```java
public boolean action(Event evt, Object arg)
{
    /* Do something in response to the event */
    return true;
}
```
The following example changes the background color when the button is pressed:

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

public class ButtonApplet extends Applet {
    static Color[] colors = {
        Color.white, Color.red, Color.blue, Color.green
    };
    int colorIndex;

    public void init () {
        Button b = new Button ("Press to Change Color");
        add (b);
        setBackground (colors[0]);
    }

    public boolean action (Event evt, Object what) {
        ++colorIndex;
        if (colorIndex >= colors.length)
            colorIndex = 0;
        setBackground (colors[colorIndex]);
        repaint ();
        return true;
    }
}
```

The `actionPerformed()` method can switch on the event’s source field:

```java
Button okButton;
Button cancelButton;
/* ... */
public void
    actionPerformed(ActionEvent evt) {
    if (evt.getSource () == okButton)
        System.out.println("ok button");
    else if (evt.getSource () == cancelButton)
        System.out.println("cancel button");
}
```
**TEXT COMPONENTS**

A *TextField* object is a box containing a single line of text. At the programmer's option, the text may be user-editable. When a text field has the input focus, it intercepts keyboard events and updates its display accordingly. It performs editing and display functions without help from the application programmer.

Text fields can be used to make applications accept text input from the user interactively.

Note that the AWT does not support an interface to the clipboard. Generally, it is not possible to cut out of text fields or paste into text fields.

The **TextField** class has four constructors:

```java
public TextField ()
public TextField (int columns)
public TextField (String text)
public TextField (String text, int columns);
```

Of these, **TextField(int)** is most the frequently used. The columns argument specifies a rough size based on the size of the font. When a **TextField** object is constructed, it is editable by default.
The following code constructs a TextField:

```
TextField tf = new TextField (80);
// argument is number of columns
```

**Class TextField**

TextField objects use the methods shown in Table D.

<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>public void setText (String text)</td>
<td>Sets the text displayed in the field</td>
</tr>
<tr>
<td>public String getText ()</td>
<td>Gets the text displayed in the field</td>
</tr>
<tr>
<td>public void setFont (Font font)</td>
<td>Sets the font used to display text</td>
</tr>
<tr>
<td>public Font getFont ()</td>
<td>Gets the font used to display text</td>
</tr>
<tr>
<td>public void select (int start, int end)</td>
<td>Selects the text between two character positions</td>
</tr>
<tr>
<td>public void selectAll ()</td>
<td>Selects all the text in the field</td>
</tr>
<tr>
<td>public void getSelectedText ()</td>
<td>Gets the selected (highlighted) text, if any</td>
</tr>
<tr>
<td>public void setForeground(Color.blue)</td>
<td>Sets the text's color</td>
</tr>
<tr>
<td>public void setBackground(Color.green)</td>
<td>Sets the background color</td>
</tr>
</tbody>
</table>

When the user presses the <ENTER> key while a TextField has the input focus, the TextField generates an action event with itself as the target.

The sample program TextInputApplet.java (see course posting) is an applet that uses a TextField to receive input.

**Read-only TextFields**

A TextField can be made read-only by calling its setEditable() method:

```
public void setEditable (boolean editable)
```
A TextField can also be queried for its is-editable attribute:

```java
public boolean isEditable ()
```

Here is an example of an applet that uses a TextField to display text, but not to receive input:

```java
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class TextFieldApplet extends Applet implements ActionListener
{
    static String[] strings = {
        "Eat", "At", "Pizza", "Coliseum"
    };
    int stringIndex = 0;
    TextField myTextField;
    public void init ()
    {
        myTextField = new TextField (strings[0], 20);
        myTextField.setEditable (false);
        add (myTextField);
        Button b = new Button ("Press Here");
        b.addActionListener (this);
        add (b);
        b.requestFocus ();
    }
    public void actionPerformed(ActionEvent evt)
    {
        ++stringIndex;
        if (stringIndex >= strings.length)
            stringIndex = 0;
        myTextField.setText (strings[stringIndex]);
    }
}
```

**Class TextArea**

A text field is limited in size to one line of text. To display several lines of text, the class TextArea must be used. TextArea objects appear as boxes containing one or more rows of text as well as horizontal and vertical scroll-bars.
Both `TextArea` and `TextField` derive from `TextComponent`, and hence share the methods listed for `TextField`. Constructing a `TextArea` differs from constructing a `TextField`, in that both rows and columns are required to specify size. The constructors are as follows:

- `public TextArea ()`
- `public TextArea (int rows, int columns)`
- `public TextArea (String text)`
- `public TextArea (String text, int rows, int columns);`

`TextArea` objects handle scrolling without assistance from the application programmer. They respond to mouse and keyboard events within their own regions. `TextArea` objects do not generate action events when the user presses `<ENTER>` as `TextField` objects do.

Figure 2 shows a sample `TextArea`.

---

**Figure 2**
A sample text area

This is a text area.
Actual display will differ.
See the sample file `TextAreaApplet.java` (see course posting) for an example of a `TextArea` in use.

### Lists

A `List` object appears as a box containing a scrollable list of text items. Using the mouse, the user may scroll the list up and down and select items on the list. List objects may be constructed to allow multiple selections at one time or only one selection at one time. Figure 3 shows a sample list.

![Figure 3: A sample list]

The `List` class has two constructors:

- `public List ()`
- `public List (int rows, boolean allowMultipleSelections)`
The first constructor leaves the number of rows unspecified and does not allow multiple selections. Both constructors create empty lists. The `addItem()` method is used to add items to the list:

```java
List list = new List();
list.addItem ("January");
list.addItem ("February");
```

In addition to `addItem()`, class `List` provides the methods listed in Table E.

<table>
<thead>
<tr>
<th>Method</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>public String getItem (int index)</td>
<td>Gets the indexed item from the list</td>
</tr>
<tr>
<td>public int countItems ()</td>
<td>Returns the number of items in the list</td>
</tr>
<tr>
<td>public void select (int index)</td>
<td>Selects (highlights) the indexed item</td>
</tr>
<tr>
<td>public int getSelectedIndex ()</td>
<td>Returns the index of the selected item, or -1 if none selected</td>
</tr>
<tr>
<td>public String getSelectedItem ()</td>
<td>Returns the text of the selected item, or null if none selected</td>
</tr>
<tr>
<td>public void delItem (int position)</td>
<td>Removes the item at the indexed position</td>
</tr>
</tbody>
</table>

For lists that allow multiple selections, the following methods are also useful:

- public int[] getSelectedIndexes ()
- public String[] getSelectedItems ()

**List Events**

When the user selects an item on a `List`, the `List` generates an `ItemEvent` object with a state change of `ItemEvent.SELECTED`. When an item becomes deselected, the `List` generates an `ItemEvent`
with a state change of ItemEvent.DESTSELECTED (These event types are meaningful only for List objects.) For example:

```java
public void itemStateChanged (ItemEvent evt)
{
    if (evt.getStateChange() == ItemEvent.SELECTED)
    {
        int index = myList.getSelectedIndex ();
        /* ...do something with the selection... */
    }
}
```

When the user double-clicks on an item, the List generates an action event. The action event’s getActionCommand() method returns a String containing the text of the item.

For a full example of a List object in use, see the sample program ListApplet.java (see course posting).

## CHOICE MENUS

A Choice object appears as a text box with a button on its right side. When the user clicks on the button, the Choice object pops up a menu that allows the user to select from a list of text items. The Choice object displays the selected item in its text box.

Creating a choice menu is similar to creating a list box. Class Choice has one constructor, which takes no arguments. Like class List, class Choice has a method addItem() for adding items to the menu:

```java
Choice choice = new Choice ();
choice.addItem ("Sunday");
choice.addItem ("Monday");
choice.addItem ("Tuesday");
```
In addition to `addItem()`, class `Choice` provides the methods listed in Table F.

<table>
<thead>
<tr>
<th>Method</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>public String getItem(int index)</td>
<td>Gets the indexed item from the list</td>
</tr>
<tr>
<td>public int countItems()</td>
<td>Returns the number of items in the list</td>
</tr>
<tr>
<td>public void select(int index)</td>
<td>Selects (highlights) the indexed item</td>
</tr>
<tr>
<td>public int getSelectedIndex()</td>
<td>Returns the index of the selected item, or -1 if none selected</td>
</tr>
<tr>
<td>public String getSelectedItem()</td>
<td>Returns the text of the selected item, or null if none selected</td>
</tr>
</tbody>
</table>

When the user selects a new item, the `Choice` object generates an `ItemEvent`. The event’s `itemStateChange()` value is `ItemEvent.SELECTED` and its `getItem()` value contains the text of the selected item.

For a full example of a `Choice` object in use, see the sample program `ChoiceApplet.java` (see course posting).

**The `PopupMenu` Class**

The `PopupMenu` class is an additional class provided with JDK 1.1. It is essentially a specialization of the `Menu` class. Objects of this class are used to provide context-sensitive menus within an applet or an application.

The class contains a `setVisible(true)` method, because a pop-up menu is not visible until the user requests it.

**CHECKBOXES**

A checkbox appears as an on/off switch with a label to the right. When the user clicks in the region of the checkbox, the on/off state of the checkbox is toggled.

Checkboxes operate either independently or as a member of a checkbox group. Within a checkbox group, only one checkbox can be in the on state at one time; enabling a checkbox in a group automatically disables all the others. This kind of checkbox is sometimes called a radio button.
Class `java.awt.Checkbox` represents a checkbox. Class `java.awt.CheckboxGroup` represents a checkbox group. Whether a checkbox operates independently or as a member of a group depends on how the `Checkbox` object is constructed:

- `public Checkbox ()`
- `public Checkbox (String label)`
- `public Checkbox (String label, CheckboxGroup group, boolean state)`

The third constructor is the most general. The first argument specifies the label of the checkbox. The second specifies the checkbox group to which this new checkbox belongs. The second argument can be null to indicate that this checkbox operates independently. The third argument specifies the initial on/off state of the checkbox to be true (meaning on). The first and second constructors create checkboxes that are initially in the off state, and operate independently.

The first line creates a `Checkbox` that is initially unchecked, while the second creates one that is checked:

```java
Checkbox cb = new Checkbox ("Gravity");

cb = new Checkbox ("Snap", null, true);
```

Class `Checkbox` provides methods for getting and setting the three attributes specified in the most general constructor:

- `public String getLabel ()`
- `public void setLabel (String label)`
- `public boolean getState ()`
- `public void setState (boolean state)`
- `public CheckboxGroup getCheckboxGroup ()`
- `public void setCheckboxGroup (CheckboxGroup group)`
The following are two examples of adding an independent Checkbox to an applet:

```java
// Initial state off...
add (new Checkbox ("This is an ungrouped checkbox"));
// Initial state on...
add (new Checkbox ("So is this one", null, true));
```

**CheckboxGroup**

Creating a CheckboxGroup is straightforward. An example of creating a checkbox group and adding its members to an applet is shown below:

```java
CheckboxGroup group = new CheckboxGroup ();
add (new Checkbox ("White", group, true));
add (new Checkbox ("Red", group, false));
add (new Checkbox ("Green", group, false));
add (new Checkbox ("Blue", group, false));
```

Checkboxes in graphical user interfaces usually specify options rather than trigger actions; even so, a Checkbox object generates an ItemEvent when the user sets its state to on.

The following code changes and then queries the checked item:

```java
group.setSelectedCheckbox (box1);
Checkbox checked = group.getSelectedCheckbox ();
```

**Scrollbars**

The List and TextArea components come with built-in scrollbars. The scrollbars of these components operate without help from the application programmer. If a new kind of scrollable object is needed, the class java.awtScrollbar can be used.
A scrollbar is both a display and a control. It displays the relative position of the current value of some variable within a range of possible values. The variable is often, but not necessarily, the horizontal or vertical position of a view within its document. A scrollbar also allows the user to change the value of the variable between the minimum and maximum. The user can:

- click on arrows to increment the value a "line" at a time
- click within a range bar to increment the value a "page" at a time
- slide a "thumb" to any point in the range

A Scrollbar object has the following attributes:

- orientation, either horizontal or vertical
- value
- visible portion (a larger visible portion means a larger thumb)
- minimum and maximum values

Class Scrollbar provides three constructors:

- public Scrollbar ()
- public Scrollbar (int orientation)
- public Scrollbar (int orientation, int initialValue, int visiblePortion, int minimumValue, int maximumValue)

The first constructor creates a vertical scrollbar. The second constructor creates either a horizontal or vertical scrollbar depending on the value of the orientation argument. Orientation is represented by an integer of either the value Scrollbar.HORIZONTAL or Scrollbar.VERTICAL. In either
case, a `Scrollbar` method must be called to set a range and value for the scrollbar. These methods appear below:

- `public int getOrientation ()`
- `public int getValue ()`
- `public void setValue (int value)`
- `public int getMinimum ()`
- `public int getMaximum ()`
- `public int getVisible ()`
- `public void setUnitIncrement (int unitIncrement)`
- `public int getUnitIncrement ()`
- `public void setBlockIncrement (int blockIncrement)`
- `public int getBlockIncrement ()`
- `public void setValues (int value, int visiblePortion,int minimumValue, int maximumValue)`

**Scrollbar Events**

A `Scrollbar` object intercepts and handles most mouse events within its region. When a `Scrollbar` changes value due to a mouse event, it generates an adjustment event with one of the id values listed in Table G.

**Table G. Scrollbar id values**

<table>
<thead>
<tr>
<th>id value</th>
<th>change to scrollbar value:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>AdjustmentEvent.UNIT_INCREMENT</code></td>
<td>Increment one &quot;line&quot; up/left</td>
</tr>
<tr>
<td><code>AdjustmentEvent.UNIT_DECREMENT</code></td>
<td>Increment one &quot;line&quot; down/right</td>
</tr>
<tr>
<td><code>AdjustmentEvent.BLOCK_INCREMENT</code></td>
<td>Increment one &quot;page&quot; up/left</td>
</tr>
<tr>
<td><code>AdjustmentEvent.BLOCK_DECREMENT</code></td>
<td>Increment one &quot;page&quot; down/right</td>
</tr>
<tr>
<td><code>AdjustmentEvent.TRACK</code></td>
<td>Set to absolute position</td>
</tr>
</tbody>
</table>

For all of these event types, the target field of the event is the `Scrollbar` object that generates it.
For most applications, there is no need to distinguish among these events. A change in the scrollbar's value usually requires updating the scrolling display to match the new value. The code below shows the form of a handleEvent() method that handles scrollbar-specific events:

```java
public void adjustmentValueChanged (AdjustmentEvent evt)
{
    switch (evt.getAdjustmentType ())
    {
        case AdjustmentEvent.UNIT_INCREMENT:
        case AdjustmentEvent.UNIT_DECREMENT:
        case AdjustmentEvent.BLOCK_INCREMENT:
        case AdjustmentEvent.BLOCK_DECREMENT:
        case AdjustmentEvent.TRACK:
            Scrollbar sb = (Scrollbar)evt.getAdjustable();
            int orientation = sb.getOrientation ();
            int value = sb.getValue ();
            /* ...update display to match value */
    }
}
```

**Layout**

The positioning of added components has yet to be discussed. What class is responsible for the layout function, and what algorithm does that class use?

First, recall that some components are containers. Containers are components that contain other components. The container is responsible for the layout of the components it contains. The programmer can control layout by controlling the order in which components are added, giving the components size guidelines, setting layout parameters and ultimately by overriding the layout logic.

Class `Container` has a `doLayout()` method. The AWT system calls a container's `doLayout()` method whenever the container's layout becomes invalid. This can occur when the container adds or removes a component, or when the container is resized. (Do not call `doLayout()` directly.) The `doLayout()` method repositions and resizes components as necessary. The programmer may override the `doLayout()` method and replace its logic with more appropriate code.
Layout Managers

Rather than embed layout logic in the Container class or any of its subclasses, the AWT designers have encapsulated layout logic into several classes called layout managers. A layout manager is a class that implements the java.awtLayoutManager interface. The LayoutManager interface provides methods that a container may call on to do layout, determine minimum size, and determine preferred size. The advantage to separating LayoutManager from Container is that it allows the programmer to plug in and use any available layout manager for any Container rather than tying layout logic to a particular Container class.

The default implementation of doLayout() checks the layout manager. If the layout manager is not null, doLayout() calls the layout manager to do layout. Otherwise, doLayout() does nothing. The methods getPreferredSize() and getMinimumSize() also delegate their work to the layout manager.

The existing layout managers cover most needs. However, if a custom layout is needed, a programmer can write his or her own layout manager or write a new doLayout() method. Writing a new layout manager is usually preferable, because it allows greater code reuse.

Class Applet uses the FlowLayout layout manager by default. To plug in a new layout manager, the Container.setLayout() method can be called:

```
public void setLayout (LayoutManager mgr)
```

The layout manager may be extracted from a container by calling Container.getLayout():

```
public LayoutManager getLayout ()
```

Some of the existing layout managers and their methods are discussed below.

Class FlowLayout

A layout manager of class FlowLayout follows three steps in laying out a container:
1. Walk the components in the order in which they were added.

2. Fit as many components as possible onto a row, based on the current sizes of the components and the width of the container window.

3. Proceed until there are no more components.

Note that `FlowLayout` never resizes the components. When `FlowLayout` is used, each component must be allocated sufficient screen size at initialization time by setting attributes such as number of rows, number of columns, text, etc.

A `FlowLayout` has three attributes:

- alignment
- horizontal gap
- vertical gap

The alignment attribute is an integer with one of the following values:

- `FlowLayout.LEFT`
- `FlowLayout.RIGHT`
- `FlowLayout.CENTER`

This attribute determines how the `FlowLayout` positions its rows.

The horizontal gap is the minimum separation between components within a row. The vertical gap applies to columns.

The default alignment for the `FlowLayout` is centered, while both gaps default to 5. This describes the layout manager all applets use by default.

The following code produces the default `FlowLayout` shown in Figure 4.

```java
public void init ()
{
    setLayout (new FlowLayout ());
    add (new Button (" 1 "));
    add (new Button (" 2 "));
    add (new Button (" 3 "));
}
```

In fact the `setLayout` method in this example is unnecessary since the Applet will use `FlowLayout` by default.
The next example produces the customized `FlowLayout` shown in Figure 5.

```java
public void init ()
{
    setLayout (new FlowLayout (FlowLayout.LEFT,
                               10, 10));
    add (new Button (" 1 "));
    add (new Button (" 2 "));
    add (new Button (" 3 "));
}
```

Figure 4
---

The default layout

```
1 2 3
```

Figure 5
---

A customized layout

```
1 2 3
```
Class GridLayout

Class java.awt.GridLayout is another implementation of LayoutManager. GridLayout does container layout based on a grid with a fixed number of cells. The number of rows and number of columns are specified as arguments to GridLayout's constructor:

```java
public GridLayout (int rows, int columns)
```

Optionally, GridLayout separates cells by a fixed point distance. The horizontal and vertical gaps may be specified as arguments to GridLayout's constructor:

```java
public GridLayout (int rows, int columns, int hgap, int vgap)
```

GridLayout traverses the components of a container in the order in which they were added, filling the grid left to right and top to bottom. Unlike FlowLayout, GridLayout resizes each component to fill its cell. GridLayout ignores components beyond the first $n$, where $n$ equals the number of rows in the grid times the number of columns. If the number of components is less than $n$, GridLayout leaves cells empty.

The following code produces the grid layout shown in Figure 6.

```java
public void init ()
{
    setLayout (new GridLayout (2, 3));
    add (new Button (" 1 "));
    add (new Button (" 2 "));
    add (new Button (" 3 "));
    add (new Button (" 4 "));
}
```
**Figure 6**
A sample grid layout

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4

Horizontal and vertical gap are optional, additional arguments to a constructor. This code produces the layout shown in Figure 7.

```java
public void init ()
{
    setLayout (new GridLayout (2, 3, 10, 5));
    add (new Button (" 1 "));
    add (new Button (" 2 "));
    add (new Button (" 3 "));
    add (new Button (" 4 "));
}
```

**Note**
The buttons are larger than the FlowLayout example since GridLayout resizes the buttons so that the first row is filled by 3 buttons, as specified in the GridLayout constructor.

**Figure 7**
Another sample grid layout

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4
LESSON 3: Abstract Windowing Toolkit Basics

Class GridBagLayout

GridBagLayout is a useful layout manager, but it is not flexible enough for most applications. GridBagLayout is a more general grid-based layout manager, offering flexible shaping of the grid and flexible positioning and sizing of the components on the grid. Using GridBagLayout is far more complicated than using GridLayout; it involves coordinating several layout parameters for each component. The basics of using GridBagLayout are presented here.

GridBagLayout takes no parameters and constructs a rectangular grid of cells based on the layout parameters of each component in the container. GridBagLayout also takes the preferred size and minimum size of each component into account. Each row and column can be of any size, depending on the size requirements of the components. Each component can occupy one or more cells.

Class GridBagLayoutConstraints encapsulates the set of layout parameters associated with a component. To use a GridBagLayout:

1. Construct the GridBagLayout object.
2. Set the GridBagLayout as the layout manager of the container.
3. Add each component to the container.
4. For each component, call GridBagLayout.setConstraints() on the layout manager, passing in a GridBagLayoutConstraints object to specify the layout parameters for that object:

   ```java
   Button b = new Button("Press");
   GridBagConstraints constraints = new GridBagConstraints();
   /* ... fill in constraints ... */
   gridbag.setConstraints(b, constraints);
   ```

GridBagLayout copies the parameters from the GridBagLayoutConstraints object. This makes the object reusable for other components.
The slayout parameters associated with a single component listed in Table H.

**Table H. Single Component Layout Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>gridx, gridy</td>
<td>Specifies the top left cell position of the component.  Cell (0,0) is the top left cell of the grid. The default is GridBagConstraints.RELATIVE, which specifies that the component occupy the next available position.</td>
</tr>
<tr>
<td>gridwidth, gridheight</td>
<td>Specifies the number of rows and columns the component occupies. The default value is one. GridBagConstraints.REMAINDER specifies that the component be last in the row (for gridwidth) or column (for gridheight). GridBagConstraints.RELATIVE specifies that the component be next to last.</td>
</tr>
<tr>
<td>fill</td>
<td>Specifies how to expand the component's size when the component's display area (area of all cells it occupies) is greater than the preferred size of the component. Possible values include: GridBagConstraints.NONE (default) GridBagConstraints.HORIZONTAL GridBagConstraints.VERTICAL GridBagConstraints.BOTH</td>
</tr>
<tr>
<td>insets</td>
<td>Specifies the amount of space to leave around the component. The type of this parameter is class java.awt.Insets, which contains top, bottom, left and right fields.</td>
</tr>
<tr>
<td>anchor</td>
<td>Specifies how to position the component within its display area when the display area is larger than the component. Possible values include: GridBagConstraints.CENTER (default) GridBagConstraints.NORTH GridBagConstraints.NORTHEAST GridBagConstraints.EAST GridBagConstraints.SOUTHEAST GridBagConstraints.SOUTH GridBagConstraints.SOUTHWEST GridBagConstraints.WEST GridBagConstraints.NORTHWEST</td>
</tr>
<tr>
<td>weightx, weighty</td>
<td>Specifies relative weight to use in distribution of row and column space among components. Default weight is zero; however, the total weight within a row or column should be non-zero.</td>
</tr>
</tbody>
</table>
The following is a sample framework for an applet using GridBagLayout:

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

public class GridBagApplet extends Applet
{
    private void addComponent (Component comp,
                        GridBagConstraints constraints)
    {
        GridBagLayout gridbag =
            (GridBagLayout) getLayout ();
        gridbag.setConstraints (comp, constraints);
        add (comp);
    }

    public void init()
    {
        setLayout (new GridBagLayout ());
        GridBagConstraints c =
            new GridBagConstraints ();
        c.fill = GridBagConstraints.BOTH;
        // Make components fill cells.
        c.weightx = 1.0;
        c.weighty = 1.0;
        // Give components some weight.
        addComponent (new Button ("1"), c);
        /* ... */
    }
}
```
Applet components can be arranged as below:

```java
// A row of four buttons.
//
addComponent (new Button ("1"), c);
addComponent (new Button ("2"), c);
c.gridwidth = GridBagConstraints.RELATIVE;
    // Next to last in row.
addComponent (new Button ("3"), c);
c.gridwidth = GridBagConstraints.REMAINDER;
    // End the row
addComponent (new Button ("4"), c);

// A row of three buttons.
// First one is two cells high.
//
c.gridwidth = 1; // reset
c.gridheight = 2;
addComponent (new Button ("5"), c);
c.gridwidth = GridBagConstraints.RELATIVE;
c.gridheight = 1;
addComponent (new Button ("6"), c);
c.gridwidth = GridBagConstraints.REMAINDER;
addComponent (new Button ("7"), c);

// One more row.
//
addComponent (new Button ("8"), c);
```

The resulting layout is shown in Figure 8:
This sample demonstrates only a small portion of GridBagLayout's capability.

**Nested Panels**

The Panel is the simplest form of AWT Container. A panel represents a rectangular section of a window, containing graphics or AWT Components. The term nested panel is almost a redundancy, since a panel exists to be nested within a window. Since a panel is a container, and a container is a component, panels may be nested within panels.

Recall that class Applet derives from class Panel. In fact, class Applet derives all of its AWT functionality from class Panel. Panel methods such as `add()`, `setLayout()` and `setBackground()` are already familiar.

Panels are used to add modularity to a GUI design. There are two general scenarios in which Panels provide a solution:

- A different layout needs to be applied to a subsection of a container.
- A user interface needs to be encapsulated for reuse.

Consider the problem of producing the layout shown in Figure 9:
The uneven rows are beyond the capability of standard layout managers.
The solution is to use nested panels, as created by the code below:

```java
public void init ()
{
    // Divide overall layout into two columns:
    // Left and right columns are nested panels.
    //
    setLayout (new GridLayout (1, 2, 10, 10));
    Panel left = new Panel ();
    Panel right = new Panel ();
    add (left);
    add (right);

    // Set up left panel with three rows.
    //
    left.setLayout (new GridLayout (3, 1, 10, 10));
    left.add (new Button ("1");
    left.add (new Button ("2");
    left.add (new Button ("3");

    // Set up right panel with four rows.
    //
    right.setLayout (new GridLayout (4, 1, 10, 10));
    right.add (new Button ("4");
    right.add (new Button ("5");
    right.add (new Button ("6");
    right.add (new Button ("7");
}
```

TIP

Since the use of panels is a little easier than using GridBagLayout, you might want to consider panels before deciding to use GridBagLayout.
Extending Class Panel

The following is from ListPanel.java (see course posting):

```java
public class ListPanel extends Panel {
    List myList;
    TextField selectedName;

    public ListPanel (int rows, int columns) {
        GridBagLayout gridbag = new GridBagLayout ();
        GridBagConstraints c = new GridBagConstraints ();
        setLayout (gridbag);

        c.fill = GridBagConstraints.BOTH;
        c.gridwidth = GridBagConstraints.REMAINDER;
        c.gridheight = 1;
        c.weightx = 1.0;
        c.weighty = 1.0;

        selectedName = new TextField (columns);
        selectedName.setEditable (false);
        selectedName.setBackground (Color.white);
        c.insets = new Insets (10, 20, 0, 20);
        gridbag.setConstraints (selectedName, c);
        add (selectedName);

        myList = new List (rows, false);
        myList.setBackground (Color.white);
        c.insets = new Insets (10, 20, 10, 20);
        gridbag.setConstraints (myList, c);
        add (myList);
    }

    public void addItem (String item) {
        myList.addItem (item);
        if (myList.countItems () == 1) {
            selectedName.setText (item);
            myList.select (0);
        }
    }
    /* ... */
}
```
Class Panel can be extended to produce custom components from existing ones. The example shown on the previous page:

- contains a List and a read-only TextField
- displays the current selection from the List in the TextField

Class ListPanel defines its own addItem() method to allow the caller to add to the List.

The finished ListPanel can be used in the same manner as any other Component. For example, it can be added to an applet:

```java
public class ListPanelApplet extends Applet
    implements ItemListener {
    static String[] names = { /* ... */ }; public void init ()
    { ListPanel lp = new ListPanel (6, 20); for (int i = 0; i < names.length; ++i) lp.addItem (names[i]); // request selection events from lp lp.addItemListener (this); add (lp); }
}
```

**Nested Panels and Events**

A Panel, as the container of one or more components, must handle events generated from those components properly. For example, the ListPanel class must respond to LIST_SELECT events from the List object it contains, by updating the selectedName field to match the current selection.

In general, a Panel (or any other component, for that matter) has a choice of actions when it receives an event. It can:

- ignore the event, allowing it to pass on (the default)
- handle the event fully
- handle the event partially and allow it to pass on

To allow an event to pass on, have the event handler method return false. According to AWT event delivery logic, an event not fully handled is next delivered to the parent, which in this case is the container of the panel.
To respond to an event, write an Event method:

```java
public void itemStateChanged
    (ItemEvent evt)
{
    if (evt.getStateChange() ==
        ItemEvent.ELECTED)
    {
        String name =
            myList.getSelectedItem();
        selectedName.setText (name);
    }
}
```

To allow the panel's container to respond to the event, implement an Event interface:

```java
public class ListPanel extends Panel
    implements ItemListener,
                ItemSelectable
{
    /* ... */
    public void addItemListener(ItemListener l)
    {
        myList.addItemListener(l);
    } /* ... */
}
```

The ListPanel class acts on LIST_SELECT events and also allows those events to pass on to the container for further processing.

Because of the way events propagate through the container structure, adding nested panels only for layout purposes does not affect event handling.

**Absolute Layout**

It is sometimes useful to do away with layout management and specify the absolute sizes and positions of components within their containers. An example of this would be an applet that positions text input fields on a standardized form. The form is an image, and the text fields must appear at specific positions on the image. To disable layout, call `setLayout()`, passing
null as its argument. There are several `Component` methods used to specify size and position:

- `public void setLocation (int x, int y)`
- `public void setSize (int width, int height)`
- `public void setSize (Dimension d)`
- `public void setBounds (int x, int y, int width, int height)`

Remember that the component must still be added to the applet using:

```java
add(b)
```
LESSON SUMMARY

In this lesson, you have learned:

- AWT component classes allow the inclusion of window objects such as buttons, text fields, and scrollbars.
- AWT layout managers, such as FlowLayout and GridLayout, handle the arrangement of components on the screen.
- Panels can be manipulated directly to create custom layouts.
- An alternative to layout management is absolute layout, in which the absolute sizes and positions of components are specified.
LESSON 3: Abstract Windowing Toolkit Basics

REVIEW QUESTIONS

1. What method is invoked to put an AWT component onto an applet?

2. Which of the AWT components can produce an action event?

3. What type of layout manager does an applet contain by default?

Answers on page 220
EXERCISE

1. Write an applet that contains a Choice menu. Populate the choice menu with the names of various items for sale.

2. Extend your applet so that the price of the currently selected menu item appears in a separate text box. (The text box should not be editable.) You may display the price in whole dollars, or in dollars and cents if you are more ambitious. Be sure to display the price of the initially selected item at startup.

3. Add a “Buy” button to your applet. Add a label that reads “Total cost:” and a TextField (non-editable) to hold the total cost. The TextField should initially read zero. Handle events so that when the user presses the “Buy” button, the total cost field increments by the price of the current menu item. The various controls and text fields might not line up properly, but don’t worry about the layout of your applet yet.

4. Add a “Clear” button to your applet. Handle events so that when the user presses the “Clear” button, the total cost field goes to zero.

5. Modify your applet to use a List rather than a Choice to display items for sale. As before, when the user selects an item, the price of the item appears in the price field. If the user double-clicks on an item, your applet should respond exactly as if the user had pressed the “Buy” button.

6. Replace your applet’s default FlowLayout manager with a layout manager that produces a more orderly appearance. BorderLayout is one possible choice. You may wish to create nested panels for organization.
OVERVIEW
The Abstract Windowing Toolkit includes features to produce standalone Java applications in addition to browser-based applets. An application usually consists of a main frame with a bordered layout, along with various menus and dialog boxes. Other components can be created and customized by extending the `Canvas` class.

LESSON TOPICS
- The Canvas Component
- Frames
- Class `BorderLayout`
- Menus
- Dialog Boxes
- Standalone Windows Applications
OBJECTIVES

By the end of this lesson, you should be able to:

- Create custom components using the Canvas class.
- Understand the concept of a frame.
- Implement a frame object using the BorderLayout class.
- Create a standalone windowing application.

THE Canvas COMPONENT

Class java.awt.Canvas exists as a means to create new components with custom graphical content. A Canvas is simply a component that can be extended and overridden. The paint() method in particular is often overridden. The programmer can control how an extension to Canvas looks and responds to events. The same is not true of the other AWT components.

For example, suppose a square button is to be created with an icon on it. No such object is provided in the AWT, but Canvas can be extended to create one. A paint() method is written for the new class. Event handlers are then created that update the appearance of the button and generate action events in response to mouse clicks. The custom button would be usable just as any other component is.
The following is a simple example. ImageComponent is a component that gets its appearance from an Image. Or in other words, ImageComponent is an image that acts as a component.

```java
import java.awt.*;

public class ImageComponent extends Canvas {
    Image myImage;

    public ImageComponent (Image image) {
        myImage = image;
    }

    public void paint (Graphics g) {
        Point loc = location ();
        Dimension sz = getSize ();
        g.drawImage (myImage, loc.x, loc.y, sz.width, sz.height, this);
    }
}
```

## Frames

A frame is an independent window that is moveable, optionally resizeable, and has a title bar and an optional menu bar. An AWT Frame, an object that represents a frame, is a type of Container, so everything that has been applied to Containers so far applies to Frames. A programmer can add Components to a Frame, set its layout manager, draw graphics in it, and so forth. Working with Frames differs from working with Applets and Panels in the following areas:

- moving, resizing, showing
- layout managers
- menus

The following code creates a Frame:

```java
Frame f = new Frame ("My Title");
```
Note that a frame is initially invisible.

The following methods are used to size, move and display a Frame:

```java
def setSize (500, 400); // width, height
f setLocation (100, 100); // x, y
f setVisible (true);
```

To next two lines hide and destroy a Frame:

```java
f setVisible (false); // hide f
f dispose (); // destroy f
```

While an applet can launch a frame, a frame is most suitable for a standalone application.

### CLASS BorderLayout

The default layout manager for a Frame object is of class `BorderLayout`. `BorderLayout`'s strategy is to arrange up to four components around the border of the frame at the north, south, east and west edges, leaving as much room as possible in the center of the frame for the main content. `BorderLayout` is well-suited for windows in which various controls, status bars, and rulers are arranged around the border of a document display.

Using `BorderLayout` requires passing an additional argument to the `add()` method when adding components to a frame. The form of `add()` to call is:

```java
public void add (String label, Component comp)
```

This form of `add()` communicates the label to the current layout manager, which associates the label with the component. Other components ignore any label passed to them, but `BorderLayout` requires a label for every
There are five labels, each of which may be assigned to at most one component:

- Center
- North
- South
- East
- West

The label specifies where in the frame the component is to appear. Because of the way `BorderLayout` works, the programmer should be aware of two things:

- If `setLayout()` is used to install a `BorderLayout` into a container, `setLayout()` must be called before `add()` is called.
- If the label argument to `add()` is omitted, the component will not appear.

`BorderLayout` arranges components as shown in Figure 10.
A different layout manager may be provided for a frame:

```
Frame f = new Frame ("Flow Layout");
f.setLayout (new FlowLayout ());
```

---

### MENUS

At the programmer’s option, a frame may have a menu bar. A menu bar is created by instantiating the class `java.awt.MenuBar`. The method `Frame.setMenuBar()` is used to add a menu to a frame. Individual menus may be added to the menu bar by instantiating class `java.awt.Menu` and passing the result to `MenuBar.add()`:

```
Frame frame = new Frame ("Untitled");
MenuBar mb = new MenuBar ();
frame.setMenuBar (mb);
Menu menu = new Menu ("File");
mb.add (menu);
```

The following items may be added to a menu:

- regular menu items (class `MenuItem`)
- checked menu items (class `CheckboxMenuItem`)
- nested menus
- separator bars

The following code adds a menu item to a menu:

```
MenuItem mi = new MenuItem ("Open");
menu.add (mi);
```

Separators can be useful in changing the look and feel of a menu:

```
menu.addSeparator ();
```
The code below works with a checked menu item:

```java
CheckboxMenuItem soundItem = new CheckboxMenuItem("Sound");
soundItem.setState(true); // initially checked
menu.add(soundItem);
/* ... */
if (soundItem.getState()) { /* ... play sound ... */ }
```

The `disable()` method is used to disable (gray out) a menu or menu item. An item is reenabled by calling its `enable()` method.

When selected, a `MenuItem` object generates an action event: the `MenuItem` is determined by the `getSource()` method.

Below is an example of handling a menu item action:

```java
public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() instanceof MenuItem) {
        MenuItem mi = (MenuItem)evt.getSource();
        String label = mi.getLabel();
        if (label.equals("New")) {
            this.new();
        } else if (label.equals("Open")) {
            this.open();
        } else if (label.equals("Exit")) {
            System.exit(0);
        }
    }
}
```
DIALOG BOXES

A dialog box is a window similar to a frame, but with the following differences:

- It may not have a menu bar.
- It may be subordinate to a frame, always displayed on top of its frame.
- It may be modal, blocking all input to its frame.

To create a Dialog, instantiate class `java.awt.Dialog` using one of the following constructors:

- `public Dialog (Frame parent, boolean modal)`
- `public Dialog (Frame parent, String title, boolean modal)`

The parent frame argument may be null, specifying that the dialog is independent of any frame.

Class `Dialog` shares the following methods with class `Frame`:

- `setLocation()`
- `setSize()`
- `setVisible(true)`
- `setVisible(false)`
- `dispose()`

If a Dialog is modal, then calling its `setVisible(true)` method causes it to appear and block all input to its frame and sibling dialogs until it is disposed of by a call to its `dispose()` method. If a Dialog is non-modal, then it may coexist with its frame and other dialogs.

The default layout manager for class `Dialog` is `BorderLayout`, the same as for class `Frame`. 
STANDALONE WINDOWS APPLICATIONS

The typical windowed application starts by putting up a main window. The rest of the program is usually an event loop, in which the program responds to a series of input events from the user until an exit event occurs. In the Java AWT, the event loop is part of the system code and is outside of the application program. The application program contains the event handlers. The system continues to process user input and post events to the proper AWT component until some component signals an exit, terminating the entire process. While there are various ways to signal a process exit, the most straightforward is to call `System.exit()` from an event handler. `System.exit()` stops all threads running in the program and releases all resources to the operating system.

A standalone application must have a `main()` method in one of its classes, declared:

```
public static void main (String[] args) {...}
```

This method typically:

- creates the main frame
- displays the main frame
- returns (which does not exit the program, because showing the frame starts independent event processing threads)

The `main()` method must be a member of some public class. It is often convenient to make `main()` a member of the extension to `Frame` that represents the main window of the application.

The following is a typical `main()`:

```
public static void main (String[] args)
{
    Frame myFrame = new MyFrame ();
    myFrame.setBounds (100, 100, 400, 200);
    myFrame.setVisible (true);
}
```
Below is a typical main frame class:

```java
public class MyFrame extends Frame {
    public MyFrame () {
        super ("My Application");
        MenuBar mb = new MenuBar ();
        setMenuBar (mb);
        /* ... set up menus ... */
        /* ... add components ... */
    }

    public void actionPerformed (ActionEvent evt) {
        if (evt.getSource () instanceof MenuItem) {
            String lab = ((MenuItem)evt.getSource ()).getLabel ();
            if (lab.equals("Exit")) {
                dispose ();
                System.exit(0);
            }
        }
    }
}
```
LESSON SUMMARY

In this lesson, you have learned:

- The `Canvas` class can be extended to create custom graphical components.
- Frames are used as main windows and have a title bar and an optional menu bar.
- The `BorderLayout` class is well suited for windows in which a central display is surrounded by various controls.
- The `main()` method of a standalone, windowed application typically creates a main frame, displays it, and returns.
REVIEW QUESTIONS

1. What must you do to a frame before it will appear on the screen?

2. You add several components to a frame, but it appears empty on the display. What is the probable cause?

3. In what ways is a dialog box different from a frame?

Answers on page 221
EXERCISE

1. Create a standalone application that behaves just as your applet in Lesson 3 does, but so that it uses a Frame rather than an Applet for its display. Add a “File” menu to your Frame. Add an “Exit” menu item to your menu. Handle events so that selecting “File...Exit” exits your program. (Hint: you can share code between applet and application by moving your applet code into a custom component class, an extension of Panel, then adding the custom component to both applet and application.)
LESSON 5

Exceptions

OVERVIEW
The use of exceptions in Java allow errors to be handled in an efficient and controlled manner. Instead of using arcane error codes or non-standard detection techniques, the programmer can use the built-in exception model to deal with errors. This lesson covers the methods used to throw and catch exceptions which will, in the process, also make the program more robust.

LESSON TOPICS
- Traditional Error Handling
- Creating an Exception Class
- The throw Statement
- Defining an Exception Handler
- The finally Statement
- Exception Types
- The throws Clause
**Objectives**

By the end of this lesson, you should be able to:

- Write Java code that handles error conditions raised by called methods.
- Write Java methods that use exceptions to return errors.
- Create exception classes.

**Traditional Error Handling**

Reporting error conditions and dealing with them when they come up is problematic in most programming languages including C. Traditional error handling often relies on special error codes returned from a function in case of error. One such example would be the Unix `read()` system call which returns zero to mean end of file. It returns -1 and sets a global error code on input error.

Often the programmer must choose between handling an error on the spot and returning it to be handled by the caller routine. Handling an error immediately can present modularity problems, while error reporting raises problems of its own. The following are a few such problems:

- There is no method supported by the language for describing errors.
- Programmer-invented methods are often obscure.
- Error-handling code pervades the program, as every site that calls a function that may return an error must do error-handling.

A common approach to error reporting and handling is to have functions that can produce errors return an integer code that identifies the error. This method too has its limitations.

**Java Exceptions**

Java provides exceptions as a mechanism for error reporting and handling. An exception is a programmer-defined condition that can be generated when appropriate and handled by user-defined code that looks for it. This allows
the programmer to decide what the problems are and what to do about them. Exceptions are application-general and independent of normal method return value. A n exception is an object that describes a failure excep-
tion, the base class being java.lang.Exception. Some exception sub-
classes are defined by the Java run-time.

The programmer may also define exception classes. When an error is
detected, the method that detects it can throw an exception. The calling
code can define an exception handler to catch the exception and respond to the
error. If an exception is not caught, it causes execution to terminate with an
error message. This process is illustrated in Figure 11.

**Creating an Exception Class**

Creating an exception class is a matter of extending the base class
java.lang.Exception. The base class provides two constructors:

- Exception ()
- Exception (String message)

The first constructs an object without a detail message while the second con-
structs an object with a detail message.
Class `Exception` also provides:

```java
   String getMessage ()
```

which the programmer may wish to override, and

```java
   void printStackTrace ()
```

which prints a trace of the calling stack to the standard error stream.

Here is an example of creating an exception class:

```java
class MyException extends Exception {
    MyException (String message) {
        super (message);
    }
}
```

Exceptions can carry a detail message, which must be set in the constructor:

```java
class MyException extends Exception {
    MyException (String message) {
        super (message);
    }
}
```

The message is exported through the `getMessage()` method:

```java
public String getMessage ()
```

THE `throw` STATEMENT

Exceptions are objects of type `Throwable` or a subclass of `Throwable`. Programmers may define their own types by deriving from `Throwable`. Code which detects any error a programmer wished to “fix” throws an excep-
tion with the syntax shown below. The thrown object must be instantiated, typically in the same statement that throws it.

```java
if (a < 10)
    throw new MyException("value is less than 10");
```

The method that executes a throw is interrupted (as if it executed a return).

### Defining an Exception Handler

To handle an exception, the code that generates the exception must be enclosed within a `try` statement.

One or more `catch` statements must follow the `try` statement, one for each type of exception being handled:

```java
try
    {
        dangerousMethod();
    }
catch (MyException e)
    {
        System.err.println (e.getMessage());
    }
```

As can be observed in the above example, there are three parts to the handling syntax:

```java
try
    {
        ...
    }
```
This designates code blocks for which the possibility of exceptions should be monitored.

```java
catch(exception-type object-name)
{
    ...
}
```

This designates code to be run if any exceptions of the designated type are “caught”.

```java
finally
{
    ...
}
```

This designates code that must be run if any part of the try block is run, typically used to clean up loose ends or close files, etc.

**The catch Statement**

If an exception is thrown, the associated catch block will catch it only if it is the appropriate type. If so, any code in the catch block will be run, then the finally block will be run, and execution will continue.

A catch statement has exactly one parameter which must be of a class type. A catch statement’s parameter matches an exception type if:

- the types are identical
- the parameter type is a superclass of the exception type
- the parameter type is an interface the exception type implements

When an exception is thrown within a try statement, the first catch statement with a matching parameter executes. If no catch statement matches, the method terminates. Only the first matching catch statement is executed. Other matching catch statements are skipped.

If the exception is not caught but the try-catch set is nested in another, any finally block is executed, and then control reverts to the enclosing catch block. If it is still not caught, the method exits and control is passed...
back to the calling method. If no method catches the exception, a default handler takes care of it. This generally results in the program terminating.

After the catch statement executes, the program continues after the try...catch block, unless the catch statement finishes by executing a return, throw, break or continue.

It is not possible to return to the point where the exception occurred.

A nother example:

```java
try {
   a[i] = 100;
   dangerousMethod();
   ...
} catch (ArrayIndexOutOfBoundsException a) {
   // ignore
} catch (MyException m) {
   return false;
} catch (Exception e) {
   System.out.println("What’s this? "+e);
   throw e;
}
```

---

**THE finally STATEMENT**

A finally statement may appear at the end of a series of catch statements. However, if a finally appears, there does not have to be a catch preceding it. A finally statement defines code which must be executed after the code in the try statement, regardless of whether that code is interrupted by an exception. It is used to handle any necessary cleanup that should not be skipped if the method exits prematurely because of an uncaught exception. This may happen, for example, when closing a network connection or a file.
If a local catch block catches an exception, the catch code runs first, then the finally block is executed. If an uncaught exception is thrown, the finally block is executed first, then the current method exits.

finally{} is always executed, even after break, continue, return, or another throw. There is one exception: it is not executed after System.exit(0); terminates the program before the finally statement.

Here is an example of finally statements:

```java
// Suppose this object must be left
// "locked" for as short a time as
// possible...
//
// obj.lock();

try {
    dangerousMethod();
}

catch (MyException m) {
    // the finally statement executes
    // before this return!
    return false;
}

finally {
    obj.unlock();
}
```

### Exception Types

The Java API defines an object of type Throwable in the package java.lang. Many different types of exceptions are derived from the base type Throwable but scattered about the 6 packages of the Java API. There are two main forms: Errors and Exceptions. The Errors are not for use by the application programmer; the Exceptions can be caught or derived from as appropriate.

The Java run-time system defines a host of Exception subclasses.
Exceptions generated by the Java run-time system are subclasses of `java.lang.RuntimeException`. These include:

- `ArithmeticException`
- `NullPointerException`
- `ClassCastException`
- `NegativeArraySizeException`
- `ArrayIndexOutOfBoundsException`
- `OutOfMemoryException`

Other predefined exception types are:

- `NoSuchMethodException`
- `InstantiationException`
- `ClassNotFoundException`
- `InterruptedException`

### THE throws CLAUSE

Any method that may throw an exception must either catch it by itself with a `try...catch` block in the method, or else declare its intention to throw that type in its declaration using the keyword `throws`:

```java
public int getVal() throws Exception {
    ...
}
public int XX() throws Ex1, Ex2 {
    ...
}
```

Likewise, if a method invokes another method that throws an exception, the invoking method must either catch the exception, or cover that type of exception in its `throws` clause.
One cause of mystifying errors is when a method calls a method that declares an exception possible but then does not take care of it or declare it as throwable. The syntax for making this declaration is to use the keyword `throws` as shown in the example above. Subclasses of `RuntimeException` are excluded from this rule.
LESSON SUMMARY

In this lesson, you have learned:

- Exceptions are objects that are thrown to signal an error.
- The syntax for handling an exception is as follows:

```java
try{...}
catch(Exception excptn){...}
finally{...}
```

- An exception is generated using the `throw` statement.
- A function must declare the types of exceptions it can throw.
1. Find the compiler error in the following code:

```java
try {
    String str = "have you found it yet?";
    str.charAt (-1);
} catch (ArrayIndexOutOfBoundsException) {}
```

2. What does the following code fragment print?

```java
char[] arr = new char [10];
try {
    arr[10] = 'A';
    System.out.println ("A B C");
} catch (RuntimeException e) {
    System.out.println ("D E F");
    throw e;
} finally {
    System.out.println ("G H I");
}
```

Answers on page 221
EXERCISE

1. Write a test program that instantiates a `String` and initializes it to some non-null value. Attempt to access a character beyond the end of the string (you may use the `charAt()` method). What happens when you run the program?

2. Modify your test program to catch the exception and print the exception's message (use the exception's `getMessage()` method).

3. Revisit your `Clock` class. Modify the `setTime()` method to throw an exception of type `BadTime` if any of the hour, minutes or seconds values are out of range. Now create the `BadTime` class. It should extend the `Exception` class and define two constructors, one with no arguments and one with a `String` argument that specifies the error message. Test your modified `setTime()` method and your new exception class.
LESSON 6
I/O Streams

OVERVIEW
In this lesson, classes InputStream and OutputStream are presented as two abstract types which serve as the sources and receivers of bytes, respectively. In addition, specialized InputStreams and OutputStreams are discussed.

LESSON TOPICS
- Introduction to I/O Streams
- Class InputStream and Class OutputStream
- File I/O
- Memory Streams and Filter Streams
- Filter Streams
- Data Input and Data Output
- Printed Output
- The LineNumInputStream Example
- Applet File I/O
OBJECTIVES

By the end of this chapter, you should be able to:

- Use standard I/O in a Java application.
- Use a variety of specialized I/O stream classes.
- Create your own specialized stream classes.
- Implement network input in a Java applet.

INTRODUCTION TO I/O STREAMS

The concept of I/O streams is common to many programming environments. In the standard Java library, an input stream is an abstract type representing a sequential source of bytes while an output stream is an abstract type representing a receiver of bytes. The basic function of the input stream type is “read one byte,” while the basic function of the output stream type is “write one byte.”

Java abstract input/output stream classes are:

- public abstract class InputStream
- public abstract class OutputStream

Since InputStream and OutputStream are abstract, you wouldn’t instantiate these classes directly. But you can use them as parameter types so that your methods will work with different derived class types.

The abstract input stream type does not specify the source of the bytes. Concrete implementations of the input stream type do read from a specified source type, such as a disk file, memory buffer or network connection. Programs that implement input based on the abstract input stream type are adaptable to any source of input. The same idea applies to output stream types.

- public class FileInputStream extends InputStream
- public class FileOutputStream extends OutputStream

The Java library provides abstract I/O stream types as described above and provides several concrete, specialized extensions to the abstract stream types as well. All these are found in the java.io package. In this lesson, I/O
stream types in the `java.io` package are covered, and network output is dealt with in brief.

**CLASS InputStream**

An object of type `java.io.InputStream` is a source of bytes. Every concrete extension to this type implements the following method:

```java
public int read() throws IOException
```

This method reads a byte of data. If the stream is positioned at the end, `read()` returns -1; otherwise, it returns the byte read. This method can throw an exception of type `IOException`, the base class for all exceptions thrown from the `java.io` library.

Class `InputStream` also provides the following methods, all implemented in terms of the basic `read()` method:

```java
//
// Reads bytes, fills the given array.
//
public int read(byte b[]) throws IOException

//
// Reads up to len bytes into array b, starting at offset
// off in the array.
//
public int read(byte b[], int off, int len) throws IOException

//
// Reads and discards n bytes.
//
public long skip(long n) throws IOException
```
InputStream also provides the following methods which do nothing in the default implementation as such. But extensions to InputStream can perform specialized tasks. InputStream can:

- close the stream, releasing any resources associated with it:

```java
public void close() throws IOException {} 
```

- return the number of bytes available for reading without blocking on I/O:

```java
public int available() throws IOException 
```

- mark the current position in the stream so that a later call to reset() will set the stream position there. ReadLimit is the maximum number of bytes the stream can read before the mark position becomes invalid:

```java
public void mark (int readlimit) 
```

- set the stream position to the last marked position:

```java
public void reset() throws IOException 
```

- return true if this type of stream supports mark/reset:

```java
public boolean markSupported() 
```

System.in is a variable of type InputStream.
CLASS OutputStream

A n object of type java.io.OutputStream is a receiver of bytes. Every concrete extension to this type implements the following method:

```java
public void write(int byte) throws IOException
```

This method writes a byte of data (the low-order byte of the integer argument). This method can throw an exception. Sample code that deals appropriately with exceptions is illustrated later on.

Class OutputStream also provides the following methods implemented in terms of the basic write() method:

- to write an entire array of bytes:

```java
public void write(byte b[]) throws IOException
```

- to write a range of a byte array:

```java
public void write(byte b[], int off, int len) throws IOException
```

OutputStream also provides the following methods, which do nothing in the default implementation as such, but whose extensions can be used for specialized tasks. The OutputStream can:

- write out any buffered data:

```java
public void flush() throws IOException
```

- close the stream, releasing any resources associated with it:

```java
public void close() throws IOException
```

System.out and System.err are variables of type OutputStream.
**USING InputStream AND OutputStream**

InputStream.read() returns a 32-bit integer. If the read succeeds, the low-order byte of the integer is the byte that was read, and the high-order bytes are zero. If the read fails, the integer value is -1. Note that -1 is a valid byte value. The programmer should test the integer for -1 before casting it to a byte. Otherwise, the error value is indistinguishable from the valid byte value of -1.

OutputStream.write() takes a 32-bit integer argument, but writes only the low-order byte.

The following is a method that copies an InputStream to an OutputStream:

```
public static boolean copy (InputStream in, OutputStream out)
{
    try {
        int ret;
        while ((ret = in.read ()) != -1) {
            out.write (ret);
        }
        return true;
    }
    catch (IOException ioe) {
        return false;
    }
}
```

It should be remembered that InputStream.read() and OutputStream.write() work with bytes, not ints.
FILE I/O

Class FileInputStream extends InputStream to provide input from a disk file.

```java
FileInputStream in =
    new FileInputStream("file.in");
```

To create a new FileInputStream and open a file by name, use the following constructor:

```java
public FileInputStream(String name)
    throws FileNotFoundException
```

FileInputStream implements read() and the other basic InputStream methods to work with files. FileInputStream implements close().

Class FileOutputStream extends OutputStream to provide output to a disk file.

```java
FileOutputStream out =
    new FileOutputStream("file.out");
```

To create a new FileOutputStream and open a file by name, use the following constructor:

```java
public FileOutputStream(String name)
    throws IOException
```

FileOutputStream implements write() and the other basic OutputStream methods to work with files. FileOutputStream implements close().

When a FileInputStream object is garbage-collected, its finalize() method executes, closing the stream. The same is true for FileOutputStream objects. However, depending on garbage collection to close streams is not a good idea for many applications, including those that
open a large number of streams and those that read and write the same file. The programmer must take care to close unused streams.

Here is an example of class FileInputStream:

```java
public static void openAndRead (String fileName, byte[] data)
{
    FileInputStream in = null;
    try {
        in = new FileInputStream (fileName);
        in.read (data);
    }
    catch (FileNotFoundException fnfe) {
        /* ... handle file open error ... */
        return;
    }
    catch (IOException ioe) {
        /* ... handle input error ... */
        /* DO NOT RETURN WITHOUT CLOSING FILE */
    }
    try {
        // Close the file.
        in.close ();
    }
    catch (IOException ioe) {
    }
}
```
Class FileOutputStream Example

```java
public static void openAndWrite(String fileName, byte[] data)
{
    FileOutputStream out;
    try {
        out = new FileOutputStream(fileName);
    }
    catch (IOException ioe) {
        /* ... handle open error ... */
        return;
    }

    try {
        out.write(data);
    }
    catch (IOException ioe) {
        /* ... handle write error ... */
        /* DO NOT RETURN WITHOUT CLOSING FILE */
    }

    try {
        out.close();
    }
    catch (IOException ioe) {
    }
}
```

In the sample method above, exception handling may seem more complicated than necessary. The complication is that the FileOutputStream constructor, write() method and close() method all throw the same type of exception—IOException—but the errors the three methods produce ought to be handled differently. If the constructor fails, the stream is not open, and the stream instance variable is not initialized. If the write() fails, the stream needs to be closed before exiting. If the close() fails, there is nothing that can be done.

### MEMORY STREAMS

Class ByteArrayInputStream extends InputStream to take input from a byte array in memory.
To create a new ByteArrayInputStream, use one of the following constructors:

- to construct a stream to read from the whole array:

  ```java
  public ByteArrayInputStream(byte[] buf)
  ```

- to construct a stream to read from only a portion of the array:

  ```java
  public ByteArrayInputStream(byte[] buf, int startOffset, int length)
  ```

Class StringBufferInputStream is similar to ByteArrayInputStream, but takes input from a String.

```java
public StringBufferInputStream(String buf)
```

Class ByteArrayOutputStream extends OutputStream to provide output to a byte array in memory. To create a new ByteArrayOutputStream, use one of the following constructors:

- public ByteArrayOutputStream()

- public ByteArrayOutputStream(int initialAllocationSize)

The ByteArrayOutputStream manages its own internal buffer. The buffer expands as needed to store the output. To extract the byte array from a ByteArrayOutputStream, call the following method:

```java
public byte[] toByteArray ()
```

The contents of the ByteArrayOutputStream can be extracted as a String by calling:

```java
public String toString ()
```
FILTER STREAMS

The stream classes described above provide only unbuffered input and output of bytes. But often the following are needed:

- buffered I/O
- unformatted I/O of basic data types (other than byte)
- formatted I/O of basic data types

The Java I/O library provides these additional functions through filter stream classes. A filter input stream is a stream that takes its input from another input stream. The class `FilterInputStream` is the base of all filter input stream classes.

**Figure 12**

An input stream pipeline

Figure 12 shows an input stream pipeline extending from a data source. The stream that reads from the data source is of a concrete `InputStream` class such as `FileInputStream`. The other streams are filter streams; each one takes its input from the next, adds special input processing, and adds methods as well.

Extensions to `FilterInputStream` provided in the `java.io` package add buffering and input of basic data types. The programmer may extend `FilterInputStream` to create specialized input stream classes.
Class `FilterInputStream` is a concrete class, but by itself it is not useful. It simply implements each `InputStream` method in terms of the corresponding input method of the attached `InputStream`.

```java
public class FilterInputStream implements InputStream
{
    InputStream attachedStream;

    public int read () throws IOException
    {
        return attachedStream.read ();
    }

    /* ... */
}
```

Subclasses of `FilterInputStream` add functionality by overriding selected methods.

`FilterInputStream`'s output counterpart is class `FilterOutputStream`. `FilterOutputStream` is the base class for filter output streams. Figure 13 shows an output stream pipeline.
Buffered I/O

BufferedInputStream is a FilterInputStream that adds buffering to any InputStream.

To add a buffer to an InputStream, use the following:

```java
static InputStream
bufferInputStream (InputStream in)
{
    return new BufferedInputStream (in);
}
```

Note that the result is just another InputStream.

The first example of a FilterInputStream is java.io.BufferedInputStream. BufferedInputStream reads from its attached InputStream a block at a time and buffers the input to satisfy the caller’s requests.

Two BufferedInputStream constructors are shown below:

- to attach a buffer of default size to the InputStream:

```java
public BufferedInputStream (InputStream in)
```

- to attach a buffer of the specified size to the InputStream:

```java
public BufferedInputStream (InputStream in, int bufsize)
```

BufferedOutputStream is a FilterOutputStream that adds buffering to any OutputStream.

The following sample illustrates how to add a buffer to an OutputStream:

```java
static OutputStream
bufferOutputStream (OutputStream out)
{
    return new BufferedOutputStream (out);
}
```
Class `BufferedOutputStream` extends `FilterOutputStream`. `BufferedOutputStream` stores its output in an internal buffer and writes the entire buffer to the attached `OutputStream` when the buffer fills. `BufferedOutputStream.flush()` (inherited from `OutputStream`) flushes the buffer to the attached `OutputStream`.

`BufferedOutputStream` constructors include:

- To attach a buffer of default size to the `OutputStream`:

  ```java
  public BufferedOutputStream (OutputStream in)
  ```

- To attach a buffer of the specified size to the `OutputStream`:

  ```java
  public BufferedOutputStream (OutputStream in, int bufsize)
  ```

### DATA INPUT

Class `DataInputStream` is a `FilterInputStream` that implements the interface `java.io.DataInput`.

Class `DataInputStream` adds methods for reading primitive types:

```java
InputStream fileIn =
    new FileInputStream("file.in");
InputStream bufIn =
    new BufferedInputStream (fileIn);
DataInputStream dataIn =
    new DataInputStream (bufIn);
boolean b = dataIn.readBoolean ();
int ub = dataIn.readUnsignedByte ();
short s = dataIn.readShort ();
int us = dataIn.readUnsignedShort ();
char c = dataIn.readChar ();
int i = dataIn.readInt ();
long l = dataIn.readLong ();
float f = dataIn.readFloat ();
double d = dataIn.readDouble ();
```

Note that this is binary input, not formatted character input.
To access these extra methods, the object is accessed as a `DataInputStream`, and not just an `InputStream`.

The `DataInput` interface defines methods for reading unformatted values of the basic types. The methods this interface defines are:

- `void readFully(byte b[])` throws `IOException`
- `void readFully(byte b[], int off, int len)` throws `IOException`
- `int skipBytes(int n)` throws `IOException`
- `boolean readBoolean()` throws `IOException`
- `byte readByte()` throws `IOException`
- `int readUnsignedByte()` throws `IOException`
- `short readShort()` throws `IOException`
- `int readUnsignedShort()` throws `IOException`
- `char readChar()` throws `IOException`
- `int readInt()` throws `IOException`
- `long readLong()` throws `IOException`
- `float readFloat()` throws `IOException`
- `double readDouble()` throws `IOException`
- `String readLine()` throws `IOException`
- `String readUTF()` throws `IOException`

The `readFully` methods are similar in function to `InputStream.read()`, while `skipBytes()` is similar to `InputStream.skip()`.

Each of the next group of methods reads a value of a primitive type. Note that `readUnsignedByte()` and `readUnsignedShort()` return 32-bit integers, as Java byte and short types are signed.

`readLine()` and `readUTF()` are similar in that each reads a line of input. `readLine()` interprets its input as ASCII, while `readUTF()` interprets its input as Unicode.
The above `DataInputStream` input methods throw `EOFException` on end of file. An `EOFException` is a kind of `IOException` which does not necessarily represent an error:

```java
// Read a file of ints.
for (;;) {
    try {
        int i = dataIn.readInt ();
        /* ... do something with i ... */
    }
    catch (EOFException) {
        break; // all done
    }
}
```

`BufferedReader` provides methods for reading one line at a time. The `String` returned does not contain an end-of-line character:

```java
BufferedReader in =
    new BufferedReader (newFileReader("file.in"));
String line;
while ((line = dataIn.readLine ()) != null)
{
    System.out.println (line);
}
in.close();
```
DATA OUTPUT

Class `DataOutputStream` is a `FilterOutputStream` that implements the interface `java.io.DataOutput`. Class `DataOutputStream` adds methods for writing primitive types:

```java
OutputStream fileOut = new FileOutputStream("file.out");
OutputStream bufOut = new BufferedOutputStream(fileOut);
DataOutputStream dataOut = new DataOutputStream(bufOut);
dataOut.writeBoolean(true);
dataOut.writeByte(0);
dataOut.writeShort(0);
dataOut.writeChar('A');
dataOut.writeInt(0);
dataOut.writeLong(0L);
dataOut.writeFloat((float) 0.0);
dataOut.writeDouble(0.0);
dataOut.close();
```

As with data input, this is binary output, not formatted character output.

To access these extra methods, the programmer accesses the object as a `DataOutputStream` and not as `OutputStream`.
The `DataOutput` interface defines methods for writing unformatted values of the basic types. The methods this interface defines are as follows:

- void `write(byte[] data)` throws IOException
- void `write(byte[] data, int offset, int length)` throws IOException
- void `writeBoolean(boolean b)` throws IOException
- void `writeByte(int v)` throws IOException
- void `writeShort(int v)` throws IOException
- void `writeChar(int v)` throws IOException
- void `writeInt(int v)` throws IOException
- void `writeLong(long v)` throws IOException
- void `writeFloat(float v)` throws IOException
- void `writeDouble(double v)` throws IOException
- void `writeBytes(String s)` throws IOException
- void `writeChars(String s)` throws IOException
- void `writeUTF(String s)` throws IOException

The first two `write()` methods above are similar in function to `OutputStream.write()`.

The next group of methods write values of the primitive Java types. Note that `writeByte()`, `writeShort()` and `writeChar()` take 32-bit integer arguments. Each method writes out the low-order byte or bytes of the integer. Whether the value is to be read as signed or unsigned is up to the application.
The last group of methods write Strings:

- `writeBytes()` writes an ASCII byte for each character in the String.
- `writeChars()` writes a 16-bit Unicode character for each character in the String.
- `writeUTF()` writes a UTF-encoded Unicode string, readable by `DataInput.readUTF()`.

`DataOutputStream` also provides methods for writing Strings:

```java
String text = "A string.";
//
// Write each character as a byte...
//
dataOut.writeBytes (text);
//
// Write each 16-bit character...
//
dataOut.writeChars (text);
//
// Write the string in a form readable
// by `DataInput.readUTF()`...
//
dataOut.writeUTF (text);
```

### PRINTED OUTPUT

Class `PrintWriter` provides the method to print text. Unlike `DataOutputStream`, `PrintWriter` prints its output in readable format. To construct a `PrintWriter`, call one of the following constructors:

```java
public PrintWriter(OutputStream out)
public PrintWriter(OutputStream out, boolean autoflush)
```
If the `autoflush` argument is true, the `PrintWriter` automatically calls `flush()` on its attached `OutputStream` after writing a newline character.

Class `PrintWriter` provides two basic methods:

- `print()` prints a value.
- `println()` prints a value followed by a newline character.

Below is an example of `PrintWriter` usage:

```java
OutputStream fileOut =
    new FileOutputStream("file.out");
OutputStream bufOut =
    new BufferedOutputStream(fileOut);
PrintWriter printOut =
    new PrintWriter(bufOut);
printOut.print("prints strings");
printOut.print('A');
```

`PrintWriter` prints any object by calling its `toString()` method:

```java
Object obj =
    new java.awt.Point(50, 60);
printOut.println(obj);
```

`System.out` and `System.err` are variables of type `PrintStream`, which is similar to `PrintWriter`. 
The various forms of `print()` and `println()` accept all the primitive types, plus `Object` and `char[]`:

- `public void print(boolean b)`
- `public void print(int i)`
- `public void print(char c)`
- `public void print(long l)`
- `public void print(float f)`
- `public void print(double d)`
- `public void print(String s)`
- `public void print(char s[])`
- `public void print(Object obj)`
- `public void println()`
- `public void println(boolean b)`
- `public void println(int i)`
- `public void println(long l)`
- `public void println(char c)`
- `public void println(float f)`
- `public void println(double d)`
- `public void println(String s)`
- `public void println(char s[])`
- `public void println(Object obj)`

`println()` with no arguments prints just a newline character. To print an `Object`, `print()` or `println()` calls the `Object`'s `toString()` method and prints the returned `String`.

`print()` and `println()` do not generate `IOExceptions`. Instead, they catch `IOException` and record whether an exception has ever
occurred over the history of the PrintWriter. The following method flushes the stream and returns true if there was ever an error on the stream:

```java
public boolean checkError()
```

THE LineNumInputStream Example

The complete source code to a class called LineNumInputStream is examined here. LineNumInputStream tracks the current line number; is identical to an existing class in the java.io package, LineNumberInputStream, but it is given a different name to avoid naming conflicts. This section serves as both an introduction to the LineNumberInputStream class and a study of how to extend filter streams for the programmer’s own purposes.

```java
import java.io.*;
public class LineNumInputStream
    extends FilterInputStream
{
    /* ... */
}
```

Initialization:

```java
private int lineNumber;
private int markedLineNumber;
// // Constructor.
// public LineNumInputStream (InputStream in)
// {
//     // init base class, attach to stream in.
//     super (in);
// }
```
The basic `read()` method:

```java
//
// Override FilterInputStream.read() to watch
// for newline characters.
//
public int read() throws IOException
{
    int c = super.read();
    if (c == '\n')
        ++lineNumber;
    return c;
}
```

A multi-byte `read()` method:

```java
// Read into an array of bytes. Must override
// this form of read() because the default calls
// the read() method of the attached stream
// directly, not our 'public int read()' method.
//
// We do not need to override read(byte[]),
// because that method calls this one by default.
//
public int read(byte b[], int off, int len) throws IOException
{
    int cc = super.read (b, off, len);
    if (cc > 0) {
        for (int i = 0; i < cc; ++i) {
            if (b[off + i] == '\n')
                ++lineNumber;
        }
    }
    return cc;
}
```
The `skip()` method:

```java
// Override the default skip() method.
// We must scan the skipped bytes for newlines.
public long skip (long n) throws IOException
{
    // Bug: casts long down to int.
    return read (new byte[(int)n]);
}
```

Other `InputStream` methods, rewritten to track line numbers:

```java
// Record the line number at the mark point.
public void mark (int readlimit)
{
    in.mark (readlimit);
    markedLineNumber = lineNumber;
}

// Restore the marked line and its line number.
public void reset () throws IOException
{
    super.reset ();
    lineNumber = markedLineNumber;
}
```

One additional method:

```java
// Get the current line count.
public int getLineNumber ()
{
    return lineNumber;
}
```

### Applet File I/O

For security reasons, an applet may not access the local file system. It is, however, often convenient for an applet to read a file for input. That is per-
missible only if the file resides on the host from which the applet was loaded and only through a URL connection.

Given a URL, an applet may open an input stream to it using the URL.openStream() method:

```java
URL url = ...;
InputStream in = url.openStream();
```

Here is an example of using the input stream:

```java
in = new BufferedInputStream (in);
DataInputStream dataIn =
    new DataInputStream (in);
String l;
while ((l = dataIn.readLine()) != null)
{
    /* ... do something with l ... */
}
```

Class URL represents a URL string. As seen in previous chapters, the Applet methods getDocumentBase() and getAppletBase() return URL objects. A convenient way to append a file name to a URL path is to construct a new URL object from the base URL and the file name:

```java
URL newURL = new URL (getDocumentBase (), "file.in");
```

Once the programmer has the correct URL, the Applet method openStream() opens the file at that URL, and a raw InputStream to the file is returned. The programmer can filter the stream as necessary for performance and data input.
In this lesson, you have learned:

- The `InputStream` class is a source of bytes.
- The `OutputStream` class is a receiver of bytes.
- Specialized streams allow data to be read into and out of sources such as files, `StringBuffers`, and `ByteArrays`.
- The `URL.openStream()` method is used to read a file into an applet.
**Review Questions**

1. What is the return type of the basic `InputStream.read()` method?

2. How does the method `DataInputStream.readShort()` report end of file?

3. What is the type of the variable `System.out`?

4. Which of the following `DataOutputStream` methods is useful for writing output to be read by a conventional text editor:
   - `writeBytes()`
   - `writeChars()`
   - `writeUTF()`

*Answers on page 221*
**Exercise**

1. Write a standalone Java application that preserves its state on exit and restores it on startup. Your application puts up a frame that includes:
   - an input textfield
   - an independent checkbox
   - a checkbox group of at least 3 checkboxes

   The user can change the state of any of these components and exit the program. When the program starts up again (in the same working directory), each component is in the state it was at the time of the previous exit.

   Use `DataInputStream` and `DataOutputStream` for I/O.

2. Create a new type of filter output stream that does simple byte encrypting. You may add a constant amount to each byte written, for example. Test your output stream, making sure that all three forms of the `write()` method do encryption. Then create a new type of filter input stream that decrypts the output of your output stream. Test your input stream, including all three forms of the `read()` method.

3. Write an applet that contains a text input field and a large text area. Whenever the user types a file name into the input field and presses return, your applet opens the file at the document URL and displays the file's contents.
LESSON 7
Java API

OVERVIEW
This lesson presents elements of the Java Application Program Interface (API). These include the clone method and library classes such as vector, stack, and hashtable. In addition, object wrapper classes that wrap values of certain types in Java such as int, long, char, float and double are detailed.

LESSON TOPICS
- Cloning an Object
- The Vector Class
- Object Wrapper Classes
- Class Stack
- Class Hashtable
OBJECTIVES

By the end of this lesson, you should be able to:

- Employ the clone method to copy objects.
- Employ Vector, Stack and Hashtable library classes to maintain collections of objects.
- Understand and apply object wrapper library classes.

CLONING AN OBJECT

To clone an object is to create another new object of the same type and value as the original object. The Java root class java.lang.Object defines a clone() method for this purpose:

```java
protected Object clone ()
        throws CloneNotSupportedException
```

This method is protected, which means that it is not exported by derived classes (even though other classes derive from Object as well). This method produces a shallow copy of any object, i.e., a new object in which the value of each instance data member is equal to the value of the corresponding member of the original object. For class-type variables, this implies that the copy refers to the same object as does the original. While the immediate result of a shallow copy is that both objects have the same value, it is not always appropriate for the two objects to share references internally. Because class Object’s clone is not always appropriate, it is not part of the interface of derived classes. It is left to the implementor of each class to decide whether the class should support cloning. If so, the implementor must declare the class as implementing the interface java.lang.Cloneable and also write a public clone() method for the class.

The Cloneable interface is empty; however, Object.clone() will throw an exception of type CloneNotSupportedException unless the class of the object to be cloned implements this interface.
If a class is cloneable, that is, it implements the `Cloneable` interface and exports a `clone()` method with the correct signature, a programmer can clone an object of that class as shown:

```java
MyClass obj1 = new MyClass();
MyClass obj2 = (MyClass) obj1.clone();
```

The cast is necessary because the method returns type `Object`, regardless of the actual type.

Objects that support the `clone()` method may be copied:

```java
MyClass obj1 = new MyClass();
MyClass obj2 = (MyClass) obj1.clone();
```

The cast to `MyClass` is necessary because `clone()` returns type `Object`. Only classes that implement the interface `Cloneable` may be cloned this way.

**MAKING A CLASS CLONEABLE**

To implement `clone()` for a class, write a `clone()` method that:

- is public
- returns `Object`
- takes no arguments
- throws no exceptions
- calls `Object.clone()`

If a shallow copy is appropriate for the class, then the implementation that follows will serve the purpose.
For a class to be cloneable, it must be declared implements Cloneable. It must define a public clone() method:

```java
public Object clone ()
{
    try
    {
        return super.clone ();
    }
    catch (CloneNotSupportedException e)
    {
        // This will happen only if the class does
        // not implement the Cloneable interface
        // as it should!
        //
        throw new InternalError ();
    }
}
```

**NOTE**

clone() may be inherited. Derived classes should override it when deemed necessary.

The base clone() method does a shallow copy. If a deep copy is necessary, the clone() method must clone class-type instance variables. For example:

```java
public Object clone ()
{
    try
    {
        MyClass obj = (MyClass) super.clone ();
        obj.array = new char [array.length];
        System.arraycopy (array, 0, obj.array, 0, array.length);
        return obj;
    }
    catch (CloneNotSupportedException e)
    {
        throw new InternalError ();
    }
}
```
THE Vector CLASS

Java arrays are of fixed size, yet dynamically resizeable arrays have many applications. The library class `java.util.Vector` represents a general dynamically resizeable array. Each element of a Vector is a variable of type `Object`, so the user can store any type of object in a Vector. As the Vector does not keep track of the actual type of each element, the user must do so, applying typecasts as and when necessary. The user has control over the values, allocation size and number of elements in a Vector.

The following sample illustrates the building of a Vector:

```java
Vector v = new Vector();
for (int i = 0; i < 12; ++i)
{
    Rectangle r = new Rectangle (i * width, 0, width, height);
    v.addElement (r);
}
```

While class `Vector` is implemented in terms of native Java arrays, it is a user-defined class with no special support from the Java language. Therefore, array syntax can not be used to work with vectors; instead, the `Vector` class makes public a number of methods for control and access.

Class `Vector` is cloneable. Cloning a vector creates another vector that refers to the same objects as the original.

The following illustrates the method of accessing Vector elements:

```java
int size = v.size ();
for (int i = 0; i < size; ++i)
{
    Object obj = v.elementAt (i);
    // cast to actual type:
    Rectangle r = (Rectangle) obj;
    ...
}
```
Other methods allow:

- insertion into a `Vector`
- the removal of an unnecessary element
- element lookup
- the growing or shrinking of a `Vector` as needed

### Object Wrapper Classes

Class `Vector` contains objects, yet it would be useful to create dynamic arrays of native type values as well. The solution is to make an object to store each native type value. For this purpose, and in general for any case in which it is useful to have an object stand for a native type value, the standard Java library provides a set of object wrapper classes. Each wrapper class wraps a certain type of value. Here is a summary of the object wrapper classes:

**Table I. Object Wrapper Classes**

<table>
<thead>
<tr>
<th>Class...</th>
<th>Wraps values of type...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>java.lang.Byte</code></td>
<td>byte</td>
</tr>
<tr>
<td><code>java.lang.Integer</code></td>
<td>int</td>
</tr>
<tr>
<td><code>java.lang.Long</code></td>
<td>long</td>
</tr>
<tr>
<td><code>java.lang.Character</code></td>
<td>char</td>
</tr>
<tr>
<td><code>java.lang.Float</code></td>
<td>float</td>
</tr>
<tr>
<td><code>java.lang.Double</code></td>
<td>double</td>
</tr>
<tr>
<td><code>java.lang.Boolean</code></td>
<td>boolean</td>
</tr>
</tbody>
</table>

An object of one of these classes may be constructed either by:

- a value of the wrapped type
- a String to be parsed

The constructor of the latter form can throw an exception of type `NumberFormatException` if the String is not formatted properly for the given type.

No object wrapper class provides any method for changing the stored value. Object wrappers can therefore be thought of as constants.
The following are examples of constructing object wrappers:

- Integer i = new Integer (10);
- Long l = new Long (0xffffffffffffffffL);
- Character c = new Character ('%');
- Boolean b = new Boolean (true);
- Float f = new Float (1.0F);
- Double d = new Double (0.000001);

The following exemplifies the construction of an object wrapper by parsing:

```java
String param = getParameter ("value");
try {
    Integer i = new Integer (param);
} catch (NumberFormatException nfe) {
    // Handle badly formatted number
}
```

**Classes Integer and Long**

Classes `java.lang.Integer` and `java.lang.Long` provide wrappers for the native integer types. Each of these classes extends the abstract class `java.lang.Number`, which requires that its concrete derived classes implement the following methods:

- public double doubleValue ()
- public float floatValue ()
- public int intValue ()
- public long longValue ()

The value wrapped within an `Integer` or `Long` can therefore be extracted as any of the four types: `double`, `float`, `int` or `long`. (Later releases of Java also include `byteValue()` and `shortValue()` in the interface.)
To extract the value of an `Integer` object `iobj`:

- \( \text{int } i = \text{iobj.intValue()}; \)
- \( \text{long } l = \text{iobj.longValue()}; \)
- \( \text{float } f = \text{iobj.floatValue()}; \)
- \( \text{double } d = \text{iobj.doubleValue()}; \)
- \( \text{String } s = \text{iobj.toString()}; \)

In addition to its instance methods, classes `Integer` and `Long` also provide static methods for working with values of their native types. Class `Long` provides the ability to include:

- `parseString` into `long`:

  ```java
  public static long parseLong(String s, int radix)
  throws NumberFormatException
  public static long parseLong(String s)
  throws NumberFormatException
  ```

- `parseString` into `Long` wrapper:

  ```java
  public static Long valueOf(String s)
  throws NumberFormatException
  public static Long valueOf(String s)
  throws NumberFormatException
  ```

- `format` `long` to `String`:

  ```java
  public static String toString(long l, int radix)
  public static String toString(long l)
  ```

Class `Integer` provides corresponding methods for the `int` type.
Classes `Integer` and `Long` also provide static constants `MIN_VALUE` and `MAX_VALUE`, which are set to the minimum and maximum values for the integer and long type, respectively:

```java
long l = ...;
if (l >= Integer.MIN_VALUE &&
    l <= Integer.MAX_VALUE) {
    int i = (int) l;
    ...
}
```

**Classes `Float` and `Double`**

Classes `java.lang.Float` and `java.lang.Double` provide wrappers for the native floating point types. Each of these classes extends the abstract class `java.lang.Number`, as do `Integer` and `Long`. The value wrapped within a `Float` or `Double` can therefore be extracted as any of the four types: `double`, `float`, `int` or `long`.

To extract the value of a `Float` object `fobj`:

- `int i = fobj.intValue ();`
- `long l = fobj.longValue ();`
- `float f = fobj.floatValue ();`
- `double d = fobj.doubleValue ();`
In addition to its instance methods, classes `Float` and `Double` also provide static methods for working with values of their native types. Class `Double` provides:

- parse String into `Double` wrapper:

  ```java
  public static Double valueOf (String s)
  throws NumberFormatException
  ```

- format `double` as `String`:

  ```java
  public static String toString (double d)
  ```

- get `double` value as raw bits:

  ```java
  public static long doubleToLongBits (double d)
  ```

- create `double` value from raw bits:

  ```java
  public static double longBitsToDouble (long l)
  ```

Class `Float` provides corresponding methods for the `float` type. (Methods for manipulating raw bits of a `float` work with `int`’s rather than `long`’s.)

Classes `Double` and `Float` also provide static constants `MIN_VALUE` and `MAX_VALUE`, which are set to the minimum and maximum values for the `double` and `float` type, respectively, and the constants `NEGATIVE_INFINITY`, `POSITIVE_INFINITY` and `NaN`.

- `Float.MIN_VALUE`
- `Float.MAX_VALUE`
- `Float.NEGATIVE_INFINITY`
- `Float.POSITIVE_INFINITY`
- `Float.NaN`
Class `java.lang.Character` provides a wrapper for the native type `char`. To extract the value from a `Character` object `cobj`:

```
c char ch = cobj.charValue ();
```

The value wrapped within a `Character` can be extracted by calling the method:

```
public char charValue ()
```

In addition to its instance methods, class `Character` also provides static methods for working with character values, similar in function to C’s `<ctype.h>` utilities. The important difference between the `<ctype.h>` utilities and the Java `Character` utilities is that the Java utilities are aware of Unicode. Class `Character` provides the ability to:

- identify the type of character:
  - `public static boolean isDigit (char ch)`
  - `public static boolean isSpace (char ch)`
  - `public static boolean isLowerCase (char ch)`
  - `public static boolean isUpperCase (char ch)`

- convert between lower and upper case characters:
  - `public static char toLowerCase (char ch)`
  - `public static char toUpperCase (char ch)`

- convert between digit character and digit value in any of a number of radices:
  - `public static int digit (char ch, int radix)`
  - `public static char forDigit (int digitVal, int radix)`
CLASS Stack

Class java.util.Stack is implemented in terms of java.util.Vector. Like a Vector, a Stack is made up of Object variables and can therefore hold elements of any class type or types.

```java
Stack s = new Stack();
try {
    for (;;) {
        String line = in.readLine();
        s.push(line);
    }
} catch (IOException ioe) {
}
if (s.empty()) {
    return;
}
try {
    for (;;) {
        String line = (String) s.pop();
        System.out.println(line);
    }
} catch (EmptyStackException ese) {
}
```

Additional methods give a Stack stack-like behavior:

- public Object push (Object obj)
- public Object peek () throws EmptyStackException
- public Object pop () throws EmptyStackException
- public boolean empty ()

CLASS Hashtable

An object of class java.util.Hashtable is a generic lookup table that maps a key object to its value. Keys and values are Objects.
Class `java.util.Hashtable` implements a hash table structure while providing a general, efficient way to look up the value corresponding to a key. In `Hashtable`, both values and keys are generic `Objects`. Neither values nor keys may be null. The most commonly used methods include:

- `public Object put (Object key, Object value)`
- `public Object get (Object key)`
- `public Object remove (Object key)`
- `public int size ()`
- `public boolean isEmpty ()`
- `public void clear ()`

Class `Hashtable` depends on the ability to produce a hash code for any key object. That ability is provided by the `hashCode()` method of class `Object`. The default implementation of this method is suitable for many classes but not for classes in which an object's value is not the same as its identity. Classes for which this is the case (such as `String`, `Integer` and others) must override the `hashCode()` method:

```java
public int hashCode ()
```

Hash tables are `Cloneable`. Cloning a `Hashtable` produces a new `Hashtable` identical to the original, referring to the same key and value objects.

The following example illustrates the use of `Strings` as keys, `Components` as values:

```java
Hashtable tab = new Hashtable ();
tab.put ("button1", new Button ());
tab.put ("button2", new Button ());
tab.put ("label1", new Label ());
...
String str = ...;
Component c = (Component) tab.get(str);
```
LESSON SUMMARY

In this lesson, you have learned:

- Objects that support the `clone()` method may be copied.
- The `java.util` package provides collection classes such as `vector`, `stack`, and `hashtable`.
- Object wrapper classes are used whenever an object must stand for a native type value.
1. Does the following method compile? Why or why not?

```java
void cloneIt ()
{
    Vector v = new Vector ();
    Vector v2 = v.clone ();
}
```

2. What is the name of the method that returns the length of a Vector?

3. What is the name of the class that provides methods for formatting double precision floating point values?

Answers on page 222
EXERCISE

1. Implement a class called `Queue` similar to `java.util.Stack` but that provides a FIFO queue of objects. Your class should export at least the following methods:

```java
public Queue ()
public void enqueue (Object obj)
public Object dequeue ()
    throws QueueEmptyException
public boolean empty ()
```

Implement your `Queue` class in terms of `java.util.Vector`.

Note that you must also create the exception class `QueueEmptyException`. Test your classes.

2. Extend your `Queue` class to create a class called `EventQueue` that provides one additional method:

```java
public Object waitForNext ()
```

This method is similar in function to `dequeue()`, except that if the queue is empty when this method is invoked, it waits until a new object enters the queue and returns it.

You will need to override the implementation of one other method. You may also need to synchronize some of your `Queue` methods. Keep in mind that class `Vector` is already thread-safe.

To test this class, write an applet or standalone program that has two threads running in it continually:

- One constantly waits for the next event in the queue.
- Another constantly enqueues events.

You may use any type of object as your event. Have the waiting thread print each event as it arrives. To make the test effective, have your enqueuing thread sleep for a random period between events (zero to 1000 milliseconds is a good range). To do that, you must learn to use the standard Java random number generator class `java.util.Random`. Use The Java Language Specification located on the Sun website to learn
to use Random.
OVERVIEW
This lesson introduces the concept of Java Beans while delineating the various interactive and interface design properties that are exhibited at design-time as well as run-time. In addition, it also explains the component development, customization, packaging and distribution aspects of Java Beans architecture.

LESSON TOPICS
- Introduction
- Java Beans Overview
- Events
- The Core Reflection API
- Properties
- Object Serialization
- JAR Files
- Introspection
OBJECTIVES

By the end of this lesson, you should be able to:

- Describe the role of Java Beans component architecture.
- Create Java Beans using common design patterns to expose properties and events.
- Handle events by implementing component interfaces and using event adaptors.
- Use the Beanbox tool to set properties, define event handlers, and serialize components.
- Package and incorporate Beans into applets and applications using JAR files.

INTRODUCTION

The creation of component-based development tools such as Visual Basic has been one of the more significant contributions to the software development process. Using this visual programming tool, the programmer is able to select a type of object or component from a tool bar, and position an instance of that object onto a window or dialog box. The programmer can then manipulate a set of properties for that object.

For example, a developer may select a button component, position an instance of the button on a window and then set the caption property to “Cancel.” Additionally, the programmer can specify what happens when an event occurs—such as when the button is clicked.

The process is visual since the developer sees the change in the object’s representation at design time. This process can occur because the development tool actually executes an instance of the component. The code for the component is contained within a dynamically linked module (VBX, O CX, DLL, etc.), and the module’s interface exposes the component’s properties and events according to a predefined standard. Thus, the component is not only executed at run-time after the application has been constructed, but it is also executed at design-time so that its attributes may be customized visually using the development tool.

TRIVIA

While the idea of components is actually an extension of Object-Oriented concepts, components have greatly increased the potential for software reuse, which was one of the objectives of Object-Orientation in the first place.
As the programmer develops the application using the development tool, the values of the properties and the code that handles the component’s events are saved within the application source code and associated data (resource) files. This process saves a great deal of the developer’s time and reduces the number of iterations required to finalize screen layout.

In addition to simple screen design, the component based development approach has provided a mechanism that enables certain software functions or classes to be reused easily and distributed to other programmers. The components do not need to be visual, and in the newer visual development tools, packaging a class into a component and making it accessible through the toolbar is a straightforward process.

There have been many advancements with component-based development approaches in recent years, and the use of such visual tools such as Smalltalk, Delphi, and Visual Basic is increasing dramatically. In order to become a significant development tool, Java must provide similar features. Fortunately, the Java Beans architecture embraces many of the strengths of these tools and delivers a solid foundation for component development, customization, packaging, and distribution.

**JAVA BEANS OVERVIEW**

Javasoft defines Java Bean as, “a reusable software component that can be manipulated visually in a builder tool.”

The Java Beans specification defines a component-based architecture for creating both visual and non-visual components in Java. The purpose of this specification is to create a standard mechanism where third party developers can package and distribute their components. In addition, Javasoft wanted to enable a number of software tool vendors to create interactive visual programming tools which facilitate the incorporation of these components into applications.

In order to accomplish these goals, the component architecture must provide facilities to allow the examination of each component’s interface—including properties, methods, and events—during customization by the component user. This process is known as introspection. The Java language (as of JDK version 1.1) provides facilities to determine a class’s properties, methods, interfaces, and events.
Once the object's elements have been exposed, the properties and object interactions can be customized to the needs of the component user. This process enables setting properties and defining event handlers. The visual development tool facilitates this process by exposing property editors and generating event handling code.

Of course, this customization process within the development tool environment would be of little use if the modifications could not be saved after customization was complete and then recalled into memory when used within the application. For this purpose, Java provides facilities for serializing an object's contents to a file and recreating components at a later time. For most cases, Java's object serialization mechanism makes this process a trivial exercise, however in the case where Java's implementation is not sufficient, the beans developer can elect to implement an alternative mechanism.

In addition to saving the modified bean attributes, Java provides a standard file format—the JAR file—for packaging and distribution of beans. This file is in a zip format and can store class files, serialized object files, images, sound files, and other resources.

To summarize, the underlying features of Beans:

- support introspection
- expose properties & events
- provide support for customization
- allow properties to be modified
- allow event handlers to be defined
- provide support for persistence
- save customized properties and event handlers for later use
- facilitate packaging of components for distribution of JAR files
Simple Java Bean

The following example shows how simply a Java Bean can be defined. The ColorBean class has a single property, BeanColor, and no events.

```java
public class ColorBean {
    Color beanColor;
    public void setBeanColor(Color clr) {
        beanColor = clr;
    }
    public Color getBeanColor() {
        return beanColor;
    }
}
```

The BeanColor property is exposed by following a simple design pattern for properties shown below:

- write: `set <property name>(<type> <parameter>)`
- read: `<type> get <property name>()`

The `setXXX()` form is used to write the value of the property, and the `getXXX()` form is used to retrieve the value. A read-only property would have only the `getXXX()` method.

One of the objectives of the Beans specification is that the Beans model should be simple to understand, so programmers may quickly begin to write Bean components and eventually move into the more complex API. Therefore, very few language features have been modified that are specific to Java Beans. While some 1.1 features are very helpful to the Beans development process, these features are not specific to beans and may be re-used elsewhere in non-beans functions.

Note that this example bean does not need to inherit from any specific object to qualify as a bean. While a visual element that is intended to be added to a container must inherit from the `Component` class, non-visual beans may be inherited from any class. They have no restrictions.
The BeanBox Tool

The BeanBox tool found in Javasoft’s Beans Development Kit provides a simple tool for testing a Java Bean within a development tool environment.

The Toolbox window, shown in Figure 14, is used to select from available beans. The beanbox window places controls onto a form. The property editor (not shown) modifies bean properties.

The BeanBox allows the programmer to select from a list of custom components (components may be added or deleted at the programmer’s discretion) and design screen layouts or test various properties using the components.
Additionally, the programmer may define event handlers for specific events which invoke methods on other objects.

Once the programmer is finished, the layout can be saved—complete with modified properties and event handlers—to a file. This file can later be incorporated into applets and applications.

**EVENTS**

An event is simply a notification that something has happened within an object. The object where the event is triggered is known as the event source. Objects that have indicated an interest in this event are known as event listeners. When an event occurs, the event source notifies event listening by invoking a method on each of the event listeners.

For example, the Button class below provides facilities to notify listener objects in the event that the button is clicked. The code fragment shows an applet which creates an instance of the Button class:

```java
class myApplet extends Applet implements ActionListener {
    public void init() {
        myButton = new Button("OK");
        myButton.addActionListener(this);
        add(myButton);
    }
    public void actionPerformed(ActionEvent actionEvt) {
        System.out.println("Button was Clicked!");
    }
    ...
}
```

The applet registers itself as an event listener by calling the `addActionListener()` method passing “this” as the listener object. In order to call this method, the applet must implement the `ActionListener` interface which has one method—`actionPerformed()`. When the button is clicked, the button will call the `actionPerformed()` method of all registered event listeners.
Events Object Interaction

An event source may have multiple event listeners. The event listener (which implements some bean-specific interface) must register with the event source. The source adds this listener to its listener list.

When the event is fired, the source object passes an Event State Object to each of its listeners by invoking an event handler method on the listener object.

An event source may have more than one event listener, therefore the event source is almost always designed to maintain a list of listeners. While Java allows a bean to define itself as having only one listener, this is generally a special case due to a specific bean attribute which may limit the number of listeners.

Figure 15 and Figure 16 provide a high-level overview of the interactions between the event source and event listener objects.

**Figure 15**
Listener registration

- **Listener**
- **Source**
- `add<Listener Type>`
  (reference to Listener)
Event State Objects

When an event source invokes an event handler on an event listener, an instance of a class derived from `EventObject` is passed into the event handler. The purpose of this object is to communicate which object is firing the event as well as transfer any data associated with the event. Event state objects are a subclass of `java.util.EventObject`. It encapsulates event information and contains a reference to the event source object:

```java
Object getSource()
```

The `getSource()` method of `EventObject` returns a reference to the object that is firing the event. This enables a single event handler object to distinguish between several event source objects where the object may be registered as an event listener.
When the subclass of EventObject is defined, the programmer may add additional data such as coordinates of the mouse click that further define the event.

```java
class NetworkMsgEvent extends EventObject {
    int msgLength;
    byte[] msgBytes;

    public NetworkMsgEvent(int msgLength, byte[] msgBytes) {
        this.msgLength = msgLength;
        this.msgBytes = msgBytes;
    }

    public int getMsgLength() { return msgLength; }
    public byte[] getMsgBytes() { return msgBytes; }
}
```

The example above shows the definition of the NetworkMsgReceivedEvent class. In addition to the elements provided within the base EventObject class, the new subclass includes information about the network message.

Note that the name of the NetworkMsgReceivedEvent class follows the Java Beans naming convention that all subclasses of EventObject end in “Event.” These naming conventions should be observed whenever possible, as the Java Beans methodology uses similar naming patterns to deduce the bean’s features.

**Event Listener Interfaces**

In order for an event listener to register itself with an event source, the event listener must have all of the appropriate event handler methods defined for that event source. The mechanism to ensure that each event listener implements all of the required methods is to define an event listener interface associated with that event source object.
Any object that registers as a listener must implement the event listener interface for that source object.

```java
public interface NetMsgListener extends EventListener {
    public void NetMsgReceived(NetworkMsgEvent evt);
    public void NetConnectionDown(EventObject evt);
}
```

The event listener interface is an extension of `java.util.EventLis-
tener`. This base interface does not require the implementation of any methods; the derived interface specifies the methods for the event.

Similar to the `EventObject` naming convention seen in the previous section, the `EventListener` paradigm uses a naming convention where the names of subclasses of `EventListener` end in “Listener.”

### Registering Event Listeners

Each event source object must maintain a list of listener objects which must be notified when an event occurs. This list can be in any form as long as listeners may register and unregister themselves by calling methods on the event source object. A simple implementation is the `Vector()` object, however the object could also be a `Hashtable` or any other list that stores objects.

The event source must implement at least two methods — one to register event listeners (add to the list) and one to unregister event listeners (remove from the list). The naming convention should be in the form shown below:

- `add<Listener Type>(<listener reference>)`
- `remove<Listener Type>(<listener reference>)`
Example:

```java
public class NetMessagesReader {
    private Vector listeners = new Vector();
    public synchronized void addNetMsgListener(NetMsgListener listener) {
        listeners.addElement(listener);
    }

    public synchronized void removeNetMsgListener(NetMsgListener listener) {
        listeners.removeElement(listener);
    }
}
```

In the example, there is an event source object that has methods for handling registration of the NetMsgListeners seen in the EventListeners section. The addNetMsgListener() method simply adds the listener object to a list of elements stored in a vector. The removeNetMsgListener() method removes the listener object.

Note that the two methods use the synchronized keyword to prevent other threads from manipulating the list of listeners while these methods are being invoke.

**Unicast Event Sources**

The typical Java Bean should support multiple event listeners, however in some circumstances, a beans developer may wish to support only one listener. In this case, the registration function should throw an exception if more than one object attempts to register as a listener. The following example shows a registration function that protects against registration of more
than one listener by throwing the `java.util.TooManyListeners` exception.

```java
public synchronized void
addUnicastListener(UnicastListener listener)
    throws java.util.TooManyListeners
{
    if (listenerCount == 1)
    {
        throw new java.util.TooManyListeners();
    }
    else
    {
        UnicastListener = listener;
        listenerCount++;
    }
}
```

This exception should also be used in the case where the number of listeners is limited. As evidenced by the name, the `TooManyListeners` exception should be used in registration functions where the number of allowed listeners is constrained by some value.
Firing the Event

When an event source fires an event, the source object uses a method to dispatch the event. The example method `notifyMsgReceived()` first creates an event state object to pass as a parameter to the event listeners.

```java
protected void notifyMsgReceived()
{
    NetworkMsgEvent evt = new NetworkMsgEvent
        (this,msgLen,msgBytes);

    Vector copy;
    synchronized (this) {
        copy = (Vector) listeners.clone();
    }

    int count = copy.size();
    for (int ii = 0; ii < count; ii++)
    {
        NetMsgListener nml =
            (NetMsgListener)copy.elementAt(ii);
        nml.NetMsgReceived(evt);
    }
}
```

The event object references the event source (`this`) and contains eventspecific information about the network message.

Once the `EventObject` is created, the code enters a synchronized block where the list of listeners is copied into a local variable. This precaution is performed because of the possibility that the list may be manipulated by another thread while looping through the list. The synchronized block prevents this from happening since no other thread may enter a synchronized section of code (`addNetMsgListener()` and `removeNetMsgListener()` are synchronized methods) while the current thread owns the object lock.

While the synchronized method as shown above is the recommended approach, it may be an example of a situation where planning for multi-threaded interactions gets out of hand. Nevertheless, making a copy of the listeners vector allows event listeners to unregister themselves within the event handler without messing up the event dispatch method’s `for` loop.
Event Adaptors

An event adaptor is an intermediary between the source and listener objects that can provide flexibility for the event listener with respect to the methods that must be implemented. The adaptor implements the `EventListener` interface specific to the event source and uses the new Reflection API to invoke methods on the target object. The result is that the object interested in being notified about events (i.e. the event listener) no longer needs to implement the `EventListener` interface at all: the adaptor simply needs to know which method to call when an event occurs.

Fortunately, the use of adaptors is primarily for the purpose of connecting objects using generated code provided by the application development tool. However, since code generated by the development tool plays a major role within the application, it is a good idea to try to understand the basic steps involved in creating an adaptor. Additionally, if a function pointer ever has to be implemented in Java, what the Reflection API has to offer will be of interest.

An additional motivation for introducing the notion of adaptors is to create a manageable mechanism for handling events of the same type from multiple object sources. For example, consider a dialog with ten buttons. In order to listen for clicks on all ten buttons without using adaptors, the `actionPerformed()` method must sort out which button was clicked by comparing the `ActionEvent.getSource()` object with each button.

Alternatively, if a generic adaptor is created, the adaptor only needs to be instructed on what method to call for each button. Using the Reflection API, the programmer could implement a generic adaptor that could be setup to route events to the proper methods by passing a reference to the button along with the name of the method passed as a string. For example, the fol-
lowing code facilitates Event Demultiplexing—another term for routing button clicks:

```java
void RegisterAdaptors()
{
    adaptor = new MyButtonAdaptor(this);

    adaptor.registerButtonHandler(button1,"actionButton1") ;
    adaptor.registerButtonHandler(button2,"actionButton2") ;
    adaptor.registerButtonHandler(button3,"actionButton3") ;
    ...
}
```

The following code illustrates this without adaptors:

```java
public void actionPerformed(ActionEvent actionEvt) {
    if (actionEvt.getSource() == button1) {
        // Do something for button1
    } else if ((actionEvt.getSource() == button2) {
        // Do something for button2
    } else ...
    ...
    // "if" statements for buttons 3 through 10
}
```

and, the following with adaptors:

```java
public void actionButton1(ActionEvent actionEvt) {
    // Do something for button1
}
public void actionButton2(ActionEvent actionEvt) {
    // Do something for button2
}

// Separate methods for buttons 3 through 10
...
THE CORE REFLECTION API

The Core Reflection API, new with version 1.1 of the JDK, consists of a few new classes and methods that enable an object’s methods, fields, and constructors to be analyzed at run-time. Additionally, Reflection enables a programmer to look up a method based on its name and invoke the method.

The code fragment below shows an example of how a method’s name contained in a String object can be used to invoke the method.

This invokes a method named `actionButton2` on MyForm object named:

```java
// Get the instance of MyForm’s class
java.lang.Class formClass = MyForm.getClass();
// Create an array of NetworkMsgEvent
Class params[] = { NetworkMsgEvent.class };
// Get a reference to the method object
// This looks a lot like a pointer to a function!
java.lang.reflect.Method actionMethod =
    formClass.getMethod("actionButton2",params);
// Set up the parameter for the actionButton2 method
Object invokeParam[] = { new NetworkMsgEvent() };
// Invoke the method
actionMethod.invoke(this,invokeParam);
```

The code executes the following steps:

1. Get a reference to the class object of the MyForm object. This instance of class is created by the Java VM when the class is loaded. Only the Java VM can create an instance, however, a reference to this instance can be obtained with the `getNewClass()` method.

2. Declare an array variable (params) that holds the classes which define the parameters of the method to be invoked. The variable contains a reference to the class of NetworkMsgEvent which is the Event State Object passed to the target object (listener).

3. Use one of the Reflection methods to obtain a reference to the method object for `actionButton2` with a NetworkMsgEvent as an argument. The resulting reference can be thought of as a function pointer, even though it is not a pointer.

4. Create an instance of the NetworkMsgEvent parameter to pass into
the event handler. (Note that the setup for the parameters to this object such as the source, message length, and message bytes have been left out.)

5. Finally, invoke the method using Reflection's invoke() method. This method takes a reference to an array of objects representing the parameters.

### Properties

A property is a named attribute of a bean object that controls the behavior of the component. Property values can be manipulated by either using the read and write methods associated with the property, or by using a property editor within a visual development tool. The development tool will call the associated methods of a property to manipulate the value.

```java
int assets;
int liabilities;
int getAssets() { return assets; }
void setAssets(int assetValue) {
    assets = assetValue;
}
int getLiabilities() { return liabilities; }
void setLiabilities(int liabilitiesValue) {
    liabilities = liabilitiesValue;
}
int getNetWorth() { return (assets - liabilities); }
```

A property is read-only if it only has a read method, and it is read/write if it has a read and a write method.

Since external access of the property is done via calling a method, property values may reference a private field within the bean, or they may be computed. For example, a bean may have two properties, Assets and Liabilities which are stored internally using variables. A third party could be called NetWorth which is a computed value based upon other fields.

This is a simple example, but it shows that the use of methods to access properties provides significant flexibility to the beans developer. Not only can property values be calculated, but actions can be performed when properties are modified. For example, changing the Color property of a control can cause the control to be repainted with the new color.
The accessor methods for a property are known as the getter, `getXXX()`, and setter, `setXXX()`, methods (from the Java Beans specification). It is important to observe Java's naming conventions for these methods.

The getter method uses the format:

```java
<property type> get<property name>(void)
```

The setter method uses the format:

```java
void set<property name> (parameter of property type)
```

### Indexed Properties

The previous examples showed simple properties with a single value. However, sometimes a property may be better presented as a series of values similar to an array. In such a situation, the developer can use an indexed property. Access to a single element in the indexed property is done by using accessor methods which specify the index of the element.

The following example shows an application indexing into the `NetworkNodes` property and printing each element:

```java
for (int ii = 0; ii < Net.getNodeCount(); ii++)
{
    System.out.println( Net.getNetworkNodes(ii) );
}
```

Note that the index specifying the element must be an `int`. This restriction may be relaxed in future versions of the SDK.

The example accessor functions that follow illustrate that it is possible to access a single element or the entire set of property values. The bottom two methods provide access to reading and writing the entire set of property values at one time by using an array.
Bound Properties

Imagine that there is a bean property whose value is important to many objects external to the bean. In this situation, it would be convenient to have a mechanism for registering to be notified when the property value changes. Java has formalized a mechanism for handling this scenario by defining a standard for supporting bound properties.

```java
// Single element accessor methods
void setNetworkNodes(int index, NetNode node);
NetNode getNetworkNodes(int index);

// Array accessor methods
void setNetworkNodes(NetNode nodes[]);
NetNode[] getNetworkNodes();
```
A bean that supports bound properties should provide registration methods for multiple PropertyChangeListener. These include:

- **Event source (property owner):**

  ```java
  addPropertyChangeListener(PropertyChangeListener x);
  removePropertyChangeListener(PropertyChangeListener x);
  ```

  The `addPropertyChangeListener` and `removePropertyChangeListener` methods above are the standard registration methods for this purpose.

- **Fire event after property state is updated:**

  ```java
  void notifyPropertyChange(oldVal,newVal,propertyName)
  {
    // Notify all property change listeners
  }
  ```

- **PropertyChangeListener Interface:**

  ```java
  public interface java.beans.PropertyChangeListener
      extends java.util.EventListener {
      public void propertyChange(PropertyChangeEvent evt);
  }
  ```

- **PropertyChangeEvent**

  ```java
  public class java.beans.PropertyChangeEvent
      extends java.util.EventObject
  {
      public PropertyChangeEvent(Object source,
                                  String propertyName,
                                  Object oldValue,
                                  Object newValue) { ... }
      ...
  }
  ```
A property change listener must implement the `propertyChange` method. When this method is called, an instance of `PropertyChangeEvent` provides the property name, the old property value as well as the new value. The source should provide notification to the listeners after the property state is updated.

### OBJECT Serialization

A fundamental element of component software is the customization of components within the development environment, then the execution of these components later in the run-time environment. The results of the customization process must be preserved or saved to some persistent state (on hard disk) and recalled into memory when necessary.

Java provides a mechanism called Object Serialization which can be used to save the state of any given object to a disk. The interface to the object serialization mechanism is a stream, so the object can also be serialized to any interface which supports streams such as a network connection or file. The serialization information is complete enough to reconstruct the object as long as the reconstructing applet or application has access to the object’s class file. The information included in the serialization stream includes an identification for the class, and its members, as well as the values of any variables within the object.

The following example code shows the saving and restoration of a string and a button object using serialization. The `ObjectOutputStream` and

---

**NOTE**

Notice that for object serialization to work, there needs to be a cooperation between readers and writers of the object. The reader has to know the exact sequence and types of objects written and it should read them in the same order.
ObjectInputStream are used to provide the fundamental serialization functions.

```java
// Save objects to file “out.ser”
String st = “Serialization Example”;
Button btn = new Button(“OK”);
FileOutputStream outF = new FileOutputStream(“out.ser”);
ObjectOutputStream outObj = new ObjectOutputStream(outF);
outObj.writeObject(st);
outObj.writeObject(btn);
outObj.flush();

// Restore objects from file “out.ser”
FileInputStream inF = new FileInputStream(“out.ser”);
ObjectInputStream inObj = new ObjectInputStream(inF);
String st = (String) inObj.readObject();
Button btn = (Button) inObj.readObject();
```

**Serializable vs. Externalizable Interfaces**

The process for enabling a simple bean to be serializable is fairly easy: the bean must simply implement the Serializable interface which contains no methods.

Serialization of certain objects may not make sense. For example, a FileInputStream is based upon a handle to an operating system file descriptor. Serialization of the FileInputStream would store the value of the handle, but when the object is recreated (perhaps on another machine), the handle may no longer be valid. Such an object should either not be serializable, or it should implement a mechanism for saving a meaningful state and reconstructing the state at a later time.

Therefore, in order to override Java’s serialization mechanism, a class may implement the Externalizable interface. This interface provides functions to read and write the object information to an object stream:

- **writeExternal()**
- **readExternal()**
The beans developer may indicate that a specific member variable should not be serialized by adding the `transient` keyword to the variable declaration.

```java
int aNumber; // Will be serialized
transient int bNumber; // Will not be serialized
```

This feature is useful for member variables that should not or cannot be serialized. For example, the `FileInputStream` does not implement `Serializable`, therefore objects which contain a reference to `FileInputStream` cannot be serializable unless the reference to the `FileInputStream` is marked as `transient`. The example above shows the two variables: one that will be saved during the serialization process, and one that will be passed over.

### Serialization Examples

The following examples show how beans may be serialized using either the `Serializable` or the `Externalizable` interface:

#### Serializable Interface:

```java
class SimpleNumberBean implements Serializable {
    int aNumber;
    public SimpleNumberBean() {}
}
```

#### Externalizable Interface:

```java
class SimpleNumberBean implements Externalizable {
    int aNumber;
    void writeExternal(ObjectOutput obj)
        throws java.io.IOException {
        obj.writeInt(aNumber);
    }
    void readExternal(ObjectInput obj)
        throws java.io.IOException {
        aNumber = obj.readInt();
    }
```

```
As can be seen in the first example, the Serializable interface is a trivial exercise. By simply implementing the Serializable interface, all the data within the object may be easily written to any stream interface.

The second example shows an implementation of the Externalizable interface. The writeExternal function takes an ObjectOutputStream stream as a parameter. The function is responsible for serializing the contents of the object—in this case the integer aNumber.

The readExternal function must re-create the values in the object from the stream. The example uses the readInt() method of the ObjectInputStream stream to restore the value to the aNumber member.

### JAR Files

Java provides a facility for packaging beans or other resource files (such as class files, icons, multimedia files, additional data or resource files) into a single compressed file known as a Java Archive File (JAR file).

The JAR file format is fully extensible and can contain class files, icons, multimedia files such as sound or movie clips, as well as bean-specific data files or resources. The JAR file uses the .zip file format (based on the PkZip utility), so the data can be compressed to save download time during applet initialization.

In addition to compression, the JAR file can be used to retrieve many files using only one HTTP transaction. Normally, each HTTP transaction requires a new socket connection and GET message sent to the HTTP server. This overhead can add up when requesting 20 to 30 class files. By placing all the class files into one JAR file, this overhead can be avoided.

Information for beans contained in a JAR file is stored in a single text file known as the manifest file. This file has text entries showing the class name for each bean. This file can also store information regarding the bean's digital signature.

Creating the JAR file is fairly simple. A command line in the following example creates a JAR file with two beans:

```cmd
C:>jar cf MyBeans SimpleBean.class ComplexBean.class
```
INTROSPECTION

In order to provide the necessary information to enable design-time customization of a bean, Java has provided a mechanism known as Introspection. This process allows the development tool to interrogate the bean to understand its methods and then deduce the properties and events of the bean by looking for certain design properties.

For a given bean, Java can use the Reflection API to determine the object’s public methods. Once this list of methods is obtained, the methods are examined to deduce properties and events by looking for basic design patterns. Therefore, a developer specifies the properties and events by adhering to the naming conventions for properties and events. The next section covers these design patterns in detail.

Introspection Design Patterns

The following samples show common design patterns fundamental to the introspection process. To expose a simple property with read and write values, the developer provides public methods to read (get) and write (set) values for the properties. Note that the return value of the get method and the parameter of the set method must be of the same type.

```java
public void setColor(Color clr);
public Color getColor();
```

A boolean property may be implemented either as a simple property using the get/set paradigm, or alternatively, an isPropertyName() method may be used. For example, the isBrown() method above provides the same function as the getBrown() method.

```java
public boolean isBrown();
```

```java
public boolean getBrown();
public boolean setBrown(boolean isBrn);
```
Indexed properties are similar to simple properties with the exception of adding a parameter to specify which element is being read or written.

```java
public Colors getColors(int index);
public void setColors(int index, Color clr);
```

Deducing the existence of an Event is done by first looking for listener registration methods (add<ListenerType> and remove<ListenerType>).

Note that registration methods that do not throw the TooManyListeners exception are assumed to be multicast. Methods that do throw this exception are determined to be unicast.

```java
public void addColorListener(ColorListener l);
public void removeColorListener(ColorListener l);
```

Once the registration methods are identified, the parameter to these methods exposes the name of the interface which must be implemented by the event listener (e.g., ColorListener). This interface must be an extension of the EventListener interface.

The methods of the interface are then assumed to be the event handlers. So for the interface at the right, a development tool would identify the colorTurnedRed and colorTurnedBrown events.

```java
interface ColorListener extends EventListener {
    public void colorTurnedRed(ColorEvent evt);
    public void colorTurnedBrown(ColorEvent evt);
}
```

**Implementing the BeanInfo Interface**

The BeanInfo interface provides a higher level of control over what bean features are discovered rather than using the reflection and introspection services. The interface provides functions to specify a bean's description and icon as well as properties, events, and methods.
A n abbreviated form of the interface is shown below:

```java
public interface BeanInfo {
    BeanDescriptor getBeanDescriptor();
    EventSetDescriptor[] getEventSetDescriptors();
    int getDefaultEventIndex();
   PropertyDescriptor[] getPropertyDescriptors();
    int getDefaultPropertyIndex();
    MethodDescriptor[] getMethodDescriptors();
    BeanInfo[] getAdditionalBeanInfo();
    java.awt.Image getIcon(int iconKind);
}
```

In order to interrogate the features of a given bean, Java first looks for a class with `BeanInfo` appended to the class name. For example, to determine the methods, properties, and events of a component named `SimpleBean`, Java first looks for a class named `SimpleBeanBeanInfo` that implements the `BeanInfo` interface. If this class cannot be found, then Java uses the reflection API and looks for the design patterns discussed earlier.

For any of the methods in the `BeanInfo` interface, returning null will cause Java to use the default reflection and introspection services. For example, in order to associate an icon with a bean, the `BeanInfo` interface must be implemented. However, the programmer can avoid the trouble of manually specifying a bean's properties or events by simply returning null for the other methods in the `BeanInfo` interface.

The Beans Development Kit facilitates this process by providing the `SimpleBeanInfo` class. This class implements all the functions in the `BeanInfo` interface by returning null. The programmer may simply override one or more of these methods to save typing effort.

Java looks for `SimpleBeanBeanInfo` when introspecting `SimpleBean`. If this class is not available, use the default introspection mechanism to create `BeanInfo` information.
BeanInfo Example

The following example shows a class MyBean that extends the SimpleBeanInfo class and overrides the getIcon and getPropertyDescriptors methods.

```java
public class MyBeanBeanInfo extends SimpleBeanInfo {
    public Image getIcon(int iconKind) {
        Image icon = null;
        if (iconKind == BeanInfo.ICON_COLOR_32x32)
            icon = loadImage("MyBean.gif");
        return icon;
    }

    public PropertyDescriptor[] getPropertyDescriptors() {
        PropertyDescriptor pd = new PropertyDescriptor("Color", Color.class);
        PropertyDescriptor(pdReturnVal[] = { pd });
        return pdReturnVal;
    }
}
```

Note that the getIcon method returns null if the iconKind is not supported. The getPropertyDescriptors method returns a descriptor for one property, Color.
Lesson Summary

In this lesson, you have learned:

➢ Java Beans provide a mechanism for the creation of software components that can be used in a visual programming environment.

➢ A bean's features include properties, events, and methods.

➢ Beans should support multiple event listeners.

➢ Java's Reflection API allows an object's methods and class information to be interrogated.

➢ Object serialization provides persistence for a bean's customized property values.
**REVIEW QUESTIONS**

1. What interface must the event listener implement to be notified of changes in the values of bound properties?

2. What file identifies which beans are contained within a JAR file?

3. What mechanism allows an event adaptor to obtain a reference to an object’s method?

4. Write the naming conventions for the following elements:
   - event state objects
   - event listener interfaces
   - boolean properties

---

Answers on page 222
In this cookbook-style exercise, you will create a bean which can be customized using the BeanBox utility.

1. Create a new class called `AddressDisplay` that extends the `java.awt.Panel` class and implements `java.io.Serializable`. When the `AddressDisplay` is constructed, add the following three labels to the component (using “add()“): `PersonNameLbl`, `StreetLbl`, and `CityLbl`. Set the default values of these labels to `PersonNameLbl`: “John Doe”, `StreetLbl`: “1 World Trade Center”, and `CityLbl`: “New York City”. You will add to this class in the remaining exercises.

2. Now add setter and getter methods to the `AddressDisplay` class for the following three properties: `PersonName`, `Street`, and `City`. All of these properties are of type `String`. Hint: You can store the value of the properties directly into the `Text` property of each Label using the `setText()` method.

3. Create a manifest file `Manifest.mf` and include the following lines:

   ```
   Manifest-Version: 1.0
   Name: AddressDisplay.class
   Java-Bean: True
   ```

4. Compile the `AddressDisplay` class and then create a JAR file adding the `AddressDisplay` class using the command:

   ```
   C:>jar cvfm Addresses.jar Manifest.mf
   AddressDisplay.class
   ```

5. Copy this file to the \BDK\jars directory of the Beans Development Kit.

6. Run the BeanBox tool using the `run.bat` file found in \BDK\BeanBox. Your bean should be available in the toolbox. Select the `AddressDisplay` bean and add it to your form. Customize the properties with your own name and address, then save the form by selecting “Menu-->Save”. Run the beanbox tool again and reload your customized control from the file you just saved.
APPENDIX A

Hypertext Markup Language (HTML)

OVERVIEW

This appendix contains a brief overview of some common HTML tags, as well as standard Netscape extensions. Many of the Netscape extensions have been submitted for inclusion in the HTML standard.
HTML History and SGML

HTML is an application of the ISO certified Standard Generalized Markup Language (SGML) standard. SGML was first published in 1988 as a standard for electronic document exchange, archival and processing. There are other de facto standards that have the same objectives, such as Adobe's Acrobat, or Microsoft's RTF (Rich Text Format).

As a subset of SGML, HTML is much simplified. SGML documents are generally more complex and programming-like than HTML documents. You can use SGML to define HTML, which allows for greater standardization and interchangeability. HTML's advantage comes from its combination of SGML tags and constructs and standard English text markup notation. HTML is therefore easy to interpret, yet powerful enough for its purpose. The HTML standard is maintained by the Internet Engineering Task Force (IETF).

The next sections list some common HTML tags, and their usage.
### Structure

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;HTML&gt;&lt;/HTML&gt;</code></td>
<td>Document type</td>
<td>Should surround whole document</td>
</tr>
<tr>
<td><code>&lt;TITLE&gt;&lt;/TITLE&gt;</code></td>
<td>Document title</td>
<td>Should be in header</td>
</tr>
<tr>
<td><code>&lt;HEAD&gt;&lt;/HEAD&gt;</code></td>
<td>Document header</td>
<td></td>
</tr>
<tr>
<td><code>&lt;BODY&gt;&lt;/BODY&gt;</code></td>
<td>Body</td>
<td></td>
</tr>
<tr>
<td><code>&lt;!-- text --&gt;</code></td>
<td>Comment</td>
<td>(not displayed by the browser)</td>
</tr>
</tbody>
</table>

### Head Elements

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;ISINDEX&gt;</code></td>
<td>Searchable Document</td>
<td>Adds search prompt, only works if document is setup for search</td>
</tr>
<tr>
<td><code>&lt;BASE href=&quot;URL&quot;&gt;</code></td>
<td>Document's base URL</td>
<td></td>
</tr>
<tr>
<td><code>&lt;LINK rel=&quot;text&quot; href=&quot;URL&quot;&gt;</code></td>
<td>Relationship</td>
<td></td>
</tr>
<tr>
<td><code>&lt;META&gt;</code></td>
<td>Meta information</td>
<td></td>
</tr>
<tr>
<td><code>&lt;NEXTID&gt;</code></td>
<td>Identifier</td>
<td>Used by HTML editors, not humans</td>
</tr>
<tr>
<td><code>&lt;ISINDEX prompt=&quot;text&quot;&gt;</code></td>
<td>Prompt text</td>
<td>Netscape extension</td>
</tr>
</tbody>
</table>
FORMATTING: BLOCKS AND SEPARATORS

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Hn&gt;&lt;/Hn&gt;</td>
<td>Headings</td>
<td>n may be a value from 1 to 6</td>
</tr>
<tr>
<td>&lt;Hn ALIGN=LEFT</td>
<td>CENTER</td>
<td>RIGHT&gt;</td>
</tr>
<tr>
<td>&lt;/Hn&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;P&gt;&lt;/P&gt;</td>
<td>Paragraph</td>
<td>Usually double line break. Closing tag is</td>
</tr>
<tr>
<td></td>
<td></td>
<td>optional.</td>
</tr>
<tr>
<td>&lt;P ALIGN=LEFT</td>
<td>CENTER</td>
<td>RIGHT&gt;</td>
</tr>
<tr>
<td>&lt;/P&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;ADDRESS&gt;&lt;/ADDRESS&gt;</td>
<td>Address field</td>
<td>Frequently in italics</td>
</tr>
<tr>
<td>&lt;BLOCKQUOTE&gt;&lt;/BLOCKQUOTE&gt;</td>
<td>Quotation</td>
<td>Usually indented</td>
</tr>
<tr>
<td>&lt;PRE&gt;&lt;/PRE&gt;</td>
<td>Preformatted</td>
<td>Display text as-is; line breaks are</td>
</tr>
<tr>
<td></td>
<td></td>
<td>retained</td>
</tr>
<tr>
<td>&lt;PRE WIDTH=n&gt;&lt;/PRE&gt;</td>
<td>Width</td>
<td>n is the number of characters</td>
</tr>
<tr>
<td>&lt;CENTER&gt;&lt;/CENTER&gt;</td>
<td>Center</td>
<td>Netscape</td>
</tr>
<tr>
<td>&lt;BR&gt;</td>
<td>Link break</td>
<td>A single line break</td>
</tr>
</tbody>
</table>

FORMATTING: PHYSICAL

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;B&gt;&lt;/B&gt;</td>
<td>Bold</td>
<td></td>
</tr>
<tr>
<td>&lt;I&gt;&lt;/I&gt;</td>
<td>Italic</td>
<td></td>
</tr>
<tr>
<td>&lt;S&gt;&lt;/S&gt;</td>
<td>Strikeout</td>
<td>Not widely implemented</td>
</tr>
<tr>
<td>&lt;U&gt;&lt;/U&gt;</td>
<td>Underline</td>
<td>Not widely implemented</td>
</tr>
<tr>
<td>&lt;TT&gt;&lt;/TT&gt;</td>
<td>Typewriter</td>
<td>Monospaced font</td>
</tr>
<tr>
<td>&lt;BLINK&gt;&lt;/BLINK&gt;</td>
<td>Blink</td>
<td>Netscape</td>
</tr>
<tr>
<td>&lt;FONT COLOR=&quot;color specs&quot; SIZE=n FACE=&quot;typeface name&quot;&gt;&lt;/FONT&gt;</td>
<td>Font size, color, typeface</td>
<td>Netscape, n=1-7; HTML 3</td>
</tr>
<tr>
<td>&lt;BASEFONT SIZE=n&gt;</td>
<td>Base font size</td>
<td>Netscape, n=1-7; default is 3</td>
</tr>
</tbody>
</table>
### List

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;LI&gt;</td>
<td>List Item</td>
<td>Use in UL, OL, MEN, and DIR</td>
</tr>
<tr>
<td>&lt;UL&gt;&lt;/UL&gt;</td>
<td>Unordered List</td>
<td>Use &lt;LI&gt; before each list item</td>
</tr>
<tr>
<td>&lt;OL&gt;&lt;/OL&gt;</td>
<td>Ordered List</td>
<td>Use &lt;LI&gt; before each list item</td>
</tr>
<tr>
<td>&lt;MENU&gt;&lt;/MENU&gt;</td>
<td>Menu List</td>
<td>Use &lt;LI&gt; before each list item</td>
</tr>
<tr>
<td>&lt;DIR&gt;&lt;/DIR&gt;</td>
<td>Directory List</td>
<td>Use &lt;LI&gt; before each list item</td>
</tr>
<tr>
<td>&lt;DT&gt;</td>
<td>Term</td>
<td>Used with Definition Lists (DL)</td>
</tr>
<tr>
<td>&lt;DD&gt;</td>
<td>Definition</td>
<td>Used with Definition Lists (DL)</td>
</tr>
<tr>
<td>&lt;DL&gt;&lt;/DL&gt;</td>
<td>Definition List</td>
<td>Use DT and DD, not LI</td>
</tr>
</tbody>
</table>

### Netscape List Extensions

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;UL TYPE=DISC</td>
<td>CIRCLE</td>
<td>SQUARE&gt;</td>
</tr>
<tr>
<td>&lt;LI TYPE=DISC</td>
<td>CIRCLE</td>
<td>SQUARE&gt;</td>
</tr>
<tr>
<td>&lt;OL TYPE=A</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>&lt;LI TYPE=A</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>&lt;OL VALUE=n&gt;</td>
<td>Initial number</td>
<td>Entire list</td>
</tr>
<tr>
<td>&lt;LI VALUE=n&gt;</td>
<td>Initial number</td>
<td>Current and following items</td>
</tr>
</tbody>
</table>
### Links

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;A HREF=&quot;URL&quot;&gt;&lt;/A&gt;</code></td>
<td>Link</td>
<td></td>
</tr>
<tr>
<td><code>&lt;A HREF=&quot;URL#label&quot;&gt;&lt;/A&gt;</code></td>
<td>Link to specific location</td>
<td>Destination is in another document</td>
</tr>
<tr>
<td><code>&lt;A HREF=&quot;#label&quot;&gt;&lt;/A&gt;</code></td>
<td>Link to specific location</td>
<td>Destination is in current document</td>
</tr>
<tr>
<td><code>&lt;A HREF=mailto:user@host&gt;</code></td>
<td>Link for mail send</td>
<td>Brings up the browser’s mail applet with address set</td>
</tr>
<tr>
<td><code>&lt;A HREF=news:newsgroup&gt;</code></td>
<td>Link to newsgroup</td>
<td>Brings up the browser’s news reader and opens that group</td>
</tr>
<tr>
<td><code>&lt;A NAME=&quot;label&quot;&gt;&lt;/A&gt;</code></td>
<td>Define location</td>
<td></td>
</tr>
</tbody>
</table>

### Images

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;IMG SRC=&quot;URL&quot; attribute&gt;</code></td>
<td>Display Image</td>
<td></td>
</tr>
<tr>
<td><code>IMG Attributes</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGN=TOP</td>
<td>BOTTOM</td>
<td>MIDDLE</td>
</tr>
<tr>
<td>ALT=&quot;text&quot;</td>
<td>Alternate text</td>
<td>Text to display if image is not displayed</td>
</tr>
<tr>
<td>BORDER=&quot;pixels&quot;</td>
<td>Pixel size of border</td>
<td></td>
</tr>
<tr>
<td>ISMAP</td>
<td>Imagemap</td>
<td>Requires a mapfile</td>
</tr>
</tbody>
</table>
### Forms

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;FORM ACTION=&quot;URL&quot;</code></td>
<td>Define form</td>
<td></td>
</tr>
<tr>
<td>METHOD=GET</td>
<td>POST&gt;<code>&lt;/FORM&gt;</code></td>
<td></td>
</tr>
<tr>
<td>Select</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;SELECT attributes&gt;</code></td>
<td>Selection field</td>
<td>Pulldown selection list</td>
</tr>
<tr>
<td>Select Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAME=&quot;value&quot;</td>
<td>Field label</td>
<td>value is the name of the input field</td>
</tr>
<tr>
<td>SIZE=n</td>
<td>#of options</td>
<td>Not implemented in all browsers</td>
</tr>
<tr>
<td>MULTIPLE</td>
<td>Multiple selections allowed</td>
<td>Can select more than one</td>
</tr>
<tr>
<td><code>&lt;OPTION&gt;</code>option text</td>
<td>Option</td>
<td>Item that can be selected</td>
</tr>
<tr>
<td><code>&lt;OPTION SELECTED&gt;</code>option text</td>
<td>Default option</td>
<td></td>
</tr>
<tr>
<td>Text Area</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;TEXTAREA attributes&gt;</code></td>
<td>Multiple line input</td>
<td></td>
</tr>
<tr>
<td><code>&lt;TEXTAREA&gt;</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Textarea Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ROWS=nCOLS=n</td>
<td>Size of box</td>
<td></td>
</tr>
<tr>
<td>NAME=&quot;value&quot;</td>
<td>Field label</td>
<td>value is the name of the input field</td>
</tr>
<tr>
<td>Input</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;INPUT attributes&gt;</code></td>
<td>Input field</td>
<td></td>
</tr>
<tr>
<td>Input Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TYPE=&quot;TEXT</td>
<td>HIDDEN</td>
<td>IMAGE</td>
</tr>
<tr>
<td>NAME=&quot;value&quot;</td>
<td>Field label</td>
<td>value is the name of the input field</td>
</tr>
<tr>
<td>CHECKED</td>
<td>Sets default to checked</td>
<td>For use with checkboxes and radioboxes</td>
</tr>
<tr>
<td>SIZE=n</td>
<td>Box size in characters</td>
<td>For use with TEXT</td>
</tr>
<tr>
<td>MAXLENGTH=n</td>
<td>Maximum length for input</td>
<td>For use with TEXT</td>
</tr>
</tbody>
</table>
## TABLES (HTML 3)

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;TABLE&gt;&lt;/TABLE&gt;</td>
<td>Define Table</td>
<td></td>
</tr>
<tr>
<td>&lt;TABLE BORDER&gt;&lt;/TABLE&gt;</td>
<td>Table Border</td>
<td>Draw borders around table</td>
</tr>
<tr>
<td>Table Row &lt;TR&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;TR attribute&gt;&lt;/TR&gt;</td>
<td>Table Row</td>
<td></td>
</tr>
<tr>
<td>TR Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGN=LEFT</td>
<td>RIGHT</td>
<td>CENTER</td>
</tr>
<tr>
<td>VALIGN=TOP</td>
<td>MIDDLE</td>
<td>BOTTOM</td>
</tr>
<tr>
<td>Table Cell&lt;TD&gt;and Header&lt;TH&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;TD attribute&gt;&lt;/TD&gt;</td>
<td>Table cell</td>
<td>Must appear within table rows</td>
</tr>
<tr>
<td>&lt;TH attribute&gt;&lt;/TH&gt;</td>
<td>Table header</td>
<td></td>
</tr>
<tr>
<td>TD and TH Attributes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALIGN=LEFT</td>
<td>RIGHT</td>
<td>CENTER</td>
</tr>
<tr>
<td>VALIGN=TOP</td>
<td>MIDDLE</td>
<td>BOTTOM</td>
</tr>
<tr>
<td>NOWRAP</td>
<td>No line breaks</td>
<td></td>
</tr>
<tr>
<td>COLSPAN=n</td>
<td>Columns to span</td>
<td></td>
</tr>
<tr>
<td>ROWSPAN=n</td>
<td>Rows to span</td>
<td></td>
</tr>
<tr>
<td>&lt;CAPTION&gt;&lt;/CAPTION&gt;</td>
<td>Table caption</td>
<td></td>
</tr>
<tr>
<td>&lt;CAPTION ALIGN=TOP</td>
<td>BOTTOM &gt;</td>
<td>Location</td>
</tr>
</tbody>
</table>

## MISCELLANEOUS NETSCAPE EXTENSIONS

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;BODY BGCOLOR=&quot;#nnn&quot;</td>
<td>Background color</td>
<td>order is red/green/blue</td>
</tr>
<tr>
<td>&lt;BODY BACKGROUND=&quot;URL&quot;</td>
<td>Background picture</td>
<td></td>
</tr>
<tr>
<td>&lt;BODY TEXT=&quot;#nnn&quot;&gt;</td>
<td>Text color</td>
<td></td>
</tr>
<tr>
<td>&lt;BODY LINK=&quot;#nnn&quot;&gt;</td>
<td>Link color</td>
<td></td>
</tr>
<tr>
<td>&lt;BODY VLINK=&quot;#nnn&quot;&gt;</td>
<td>Visited link</td>
<td></td>
</tr>
<tr>
<td>&lt;BODY ALINK=&quot;#nnn&quot;&gt;</td>
<td>Active link</td>
<td></td>
</tr>
<tr>
<td>&lt;BR CLEAR=LEFT</td>
<td>RIGHT</td>
<td>ALL&gt;</td>
</tr>
<tr>
<td>&lt;NOBR&gt;</td>
<td>No break</td>
<td>Prevents line breaks</td>
</tr>
<tr>
<td>&lt;WBR&gt;</td>
<td>Word break</td>
<td>Where line may be broken</td>
</tr>
</tbody>
</table>
### JAVA APPLETS

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;APPLET attributes&gt;&lt;/APPLET&gt;</td>
<td>Run Java applet</td>
<td></td>
</tr>
<tr>
<td>ALIGN=’left’</td>
<td>“right”</td>
<td>“top”</td>
</tr>
<tr>
<td>ALT=’string’</td>
<td>Default text</td>
<td>Text to show if browser doesn't support Java</td>
</tr>
<tr>
<td>CODE=’URL’</td>
<td>Applet program</td>
<td>Applet program, ending with “.class”</td>
</tr>
<tr>
<td>CODEBASE=’URL’</td>
<td>Applet location</td>
<td>Needed if .class file not found in same directory as URL</td>
</tr>
<tr>
<td>HEIGHT=integer</td>
<td>Applet pixel height</td>
<td></td>
</tr>
<tr>
<td>WIDTH=integer</td>
<td>Applet pixel width</td>
<td></td>
</tr>
<tr>
<td>&lt;PARAM attributes&gt;&lt;/PARAM&gt;</td>
<td>Param Attributes</td>
<td></td>
</tr>
<tr>
<td>NAME=’parameter name’</td>
<td>Applet param name</td>
<td>Applet will ask for the value for the param of this name</td>
</tr>
<tr>
<td>VALUE=’parameter value’</td>
<td>Applet param value</td>
<td>This value is always specified and returned to program as a string</td>
</tr>
</tbody>
</table>

### JAVA SCRIPT

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;SCRIPT attributes&gt;&lt;/SCRIPT&gt;</td>
<td>Run Script</td>
<td>Designed to support new scripts in the future</td>
</tr>
<tr>
<td>LANGUAGE=’JavaScript’</td>
<td>Specifies which language is being used</td>
<td>Tells browser that the language code that follows is Java Script</td>
</tr>
<tr>
<td>function func_name(form)</td>
<td>Script</td>
<td>The actual script is embedded at this location in the HTML document</td>
</tr>
</tbody>
</table>
APPENDIX B
Java Sample

OVERVIEW
This appendix contains a sample Java applet. It comes from the demos supplied with the Java Development Kit. The applet can be viewed using any java-capable browser or using the Applet Viewer, which is part of the JDK release. To run this applet with the Applet Viewer, just specify the URL with the appletviewer command. See the course postings for a downloadable version of the applet.
Copyright (c) 1994-1996 Sun Microsystems, Inc. All Rights Reserved.

Sun grants you ("Licensee") a non-exclusive, royalty free, license to use, modify and redistribute this software in source and binary code form, provided that i) this copyright notice and license appear on all copies of the software; and ii) Licensee does not utilize the software in a manner which is disparaging to Sun.

This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This software is not designed or intended for use in on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or in the design, construction, operation or maintenance of any nuclear facility. Licensee represents and warrants that it will not use or redistribute the Software for such purposes.
THE XYZApp.JAVA SOURCE

/*
 * @(#)XYZApp.java1.3 96/12/06
 * A set of classes to parse, represent and display Chemical compounds in
 * .xyz format (see http://chem.leeds.ac.uk/Project/MIME.html)
 */
import java.applet.Applet;
import java.awt.Image;
import java.awt.Event;
import java.awt.Graphics;
import java.awt.Dimension;
import java.io.StreamTokenizer;
import java.io.InputStream;
import java.io.BufferedInputStream;
import java.io.IOException;
import java.net.URL;
import java.util.Hashtable;
import java.awt.image.IndexColorModel;
import java.awt.image.ColorModel;
import java.awt.image.MemoryImageSource;
/** The representation of a Chemical .xyz model */
class XYZChemModel {
    float vert[];
    Atom atoms[];
    int tvert[];
    int ZsortMap[];
    int nvert, maxvert;
    static Hashtable atomTable = new Hashtable();
    static Atom defaultAtom;
    static {
        atomTable.put("c", new Atom(0, 0, 0));
        atomTable.put("h", new Atom(210, 210, 210));
        atomTable.put("n", new Atom(0, 0, 255));
        atomTable.put("o", new Atom(255, 0, 0));
        atomTable.put("p", new Atom(255, 0, 255));
        atomTable.put("s", new Atom(255, 255, 0));
        atomTable.put("hn", new Atom(150, 255, 150)); /* !!*/
        defaultAtom = new Atom(255, 100, 200);
    }
    boolean transformed;
    Matrix3D mat;
    float xmin, xmax, ymin, ymax, zmin, zmax;
    XYZChemModel () {
        mat = new Matrix3D();
        mat.xrot(20);
        mat.yrot(30);
    }
}
/** Create a Chemical model by parsing an input stream */
XYZChemModel (InputStream is) throws Exception
{
    this();
    StreamTokenizer st;
    st = new StreamTokenizer(new BufferedInputStream(is, 4000));
    st.eolIsSignificant(true);
    st.commentChar('#');
    int slot = 0;
    try
    {
        scan:
        while (true)
        {
            switch ( st.nextToken() )
            {
                case StreamTokenizer.TT_EOF:
                    break scan;
                default:
                    break;
                case StreamTokenizer.TT_WORD:
                    String name = st.sval;
                    double x = 0, y = 0, z = 0;
                    if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                    {
                        x = st.nval;
                        if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                        {
                            y = st.nval;
                            if (st.nextToken() == StreamTokenizer.TT_NUMBER)
                            {
                                z = st.nval;
                            }
                        }
                    }
                    addVert(name, (float) x, (float) y, (float) z);
                    while( st.ttype != StreamTokenizer.TT_EOL &&
                        st.ttype != StreamTokenizer.TT_EOF )
                    { st.nextToken();
                    }
                    // end Switch
                    // end while
                is.close();
            } // end Try
            catch( IOException e) {}
            if (st.ttype != StreamTokenizer.TT_EOF)
            throw new Exception(st.toString());
        } // end XYZChemModel()
        /** Add a vertex to this model */
        int addVert(String name, float x, float y, float z) {
            int i = nvert;
            if (i >= maxvert)
if (vert == null) {
    maxvert = 100;
    vert = new float[maxvert * 3];
    atoms = new Atom[maxvert];
} else {
    maxvert *= 2;
    float nv[] = new float[maxvert * 3];
    System.arraycopy(vert, 0, nv, 0, vert.length);
    vert = nv;
    Atom na[] = new Atom[maxvert];
    System.arraycopy(atoms, 0, na, 0, atoms.length);
    atoms = na;
}

Atom a = (Atom) atomTable.get(name.toLowerCase());
if (a == null) a = defaultAtom;
atoms[i] = a;
i *= 3;
vert[i] = x;
vert[i + 1] = y;
vert[i + 2] = z;
return nvert++;

/** Transform all the points in this model */
void transform() {
    if (transformed || nvert <= 0)
        return;

    if (tvert == null || tvert.length < nvert * 3)
        tvert = new int[nvert * 3];
    mat.transform(vert, tvert, nvert);
    transformed = true;
}

/** Paint this model to a graphics context. It uses the matrix associated
    with this model to map from model space to screen space.
The next version of the browser should have double buffering,
which will make this *much* nicer */
void paint(Graphics g) {
    if (vert == null || nvert <= 0)
        return;
    transform();
    int v[] = tvert;
    int zs[] = ZsortMap;
    if (zs == null) {
        ZsortMap = zs = new int[nvert];
        for (int i = nvert; --i >= 0;)
            zs[i] = i * 3;
    }
    
    /*
    * I use a bubble sort since from one iteration to the next, the sort
    * order is pretty stable, so I just use what I had last time as a
    * "guess" of the sorted order. With luck, this reduces O(N log N)
* to O(N)
*/
for (int i = nvert - 1; --i >= 0;) {
    boolean flipped = false;
    for (int j = 0; j <= i; j++) {
        int a = zs[j];
        int b = zs[j + 1];
        if (v[a + 2] > v[b + 2]) {
            zs[j + 1] = a;
            zs[j] = b;
            flipped = true;
        }
    }
    if (!flipped)
        break;
}
int lg = 0;
int lim = nvert;
Atom ls[] = atoms;
if (lim <= 0 || nvert <= 0)
    return;
for (int i = 0; i < lim; i++) {
    int j = zs[i];
    int grey = v[j + 2];
    if (grey < 0)
        grey = 0;
    if (grey > 15)
        grey = 15;
    // g.drawString(names[i], v[j], v[j + 1]);
    atoms[j/3].paint(g, v[j], v[j + 1], grey);
    // g.drawImage(iBall, v[j] - (iBall.width >> 1), v[j + 1] -
    // (iBall.height >> 1));
}
/** Find the bounding box of this model */
void findBB() {
    if (nvert <= 0)
        return;
    float v[] = vert;
    float xmin = v[0], xmax = xmin;
    float ymin = v[1], ymax = ymin;
    float zmin = v[2], zmax = zmin;
    for (int i = nvert * 3; (i -= 3) > 0;) {
        float x = v[i];
        if (x < xmin)
            xmin = x;
        if (x > xmax)
            xmax = x;
        float y = v[i + 1];
        if (y < ymin)
```java
float y = ymin;
if (y > ymax)
    ymax = y;
float z = v[i + 2];
if (z < zmin)
    zmin = z;
if (z > zmax)
    zmax = z;
}
this.xmax = xmax;
this.xmin = xmin;
this.ymax = ymax;
this.ymin = ymin;
this.zmax = zmax;
this.zmin = zmin;
}
/** An applet to put a Cehmical model into a page */
public class XYZApp extends Applet implements Runnable {
    XYZChemModel md;
    boolean painted = true;
    float xfac;
    int prevx, prevy;
    float xtheta, ytheta;
    float scalefudge = 1;
    Matrix3D amat = new Matrix3D(), tmat = new Matrix3D();
    String mdname = null;
    String message = null;
    Image backBuffer;
    Graphics backGC;
    Dimension backSize;
    private synchronized void newBackBuffer() {
        backBuffer = createImage(size().width, size().height);
        backGC = backBuffer.getGraphics();
        backSize = size();
    }
    public void init() {
        mdname = getParameter("model");
        try {
            scalefudge = Float.valueOf(getParameter("scale")).floatValue();
        } catch(Exception e) {
        };
        amat.yrot(20);
        amat.xrot(20);
        if (mdname == null)
            mdname = “model.obj”;
        resize(size().width <= 20 ? 400 : size().width,
                size().height <= 20 ? 400 : size().height);
        newBackBuffer();
    }
```
public void run() {
    InputStream is = null;
    try {
        Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
        is = new URL(getDocumentBase(), mdname).openStream();
        XYZChemModel m = new XYZChemModel(is);
        Atom.setApplet(this);
        md = m;
        m.findBB();
        float xw = m.xmax - m.xmin;
        float yw = m.ymax - m.ymin;
        float zw = m.zmax - m.zmin;
        if (yw > xw)
            xw = yw;
        if (zw > xw)
            xw = zw;
        float f1 = size().width / xw;
        float f2 = size().height / xw;
        xfac = 0.7f * (f1 < f2 ? f1 : f2) * scalefudge;
    } catch(Exception e) {
        e.printStackTrace();
        md = null;
        message = e.toString();
    }
    try {
        if (is != null)
            is.close();
    } catch(Exception e) {
    }
    repaint();
}
public void start() {
    if (md == null && message == null)
        new Thread(this).start();
}
public void stop() {
}
public boolean mouseDown(Event e, int x, int y) {
    prevx = x;
    prevy = y;
    return true;
}
public boolean mouseDrag(Event e, int x, int y) {
    tmat.unit();
    float xtheta = (prevy - y) * (360.0f / size().width);
    float ytheta = (x - prevx) * (360.0f / size().height);
    tmat.xrot(xtheta);
    tmat.yrot(ytheta);
    amat.mult(tmat);
    if (painted) {
```java
    painted = false;
    repaint();
}
prevx = x;
prevy = y;
return true;
}
public void update(Graphics g) {
    if (backBuffer == null)
        g.clearRect(0, 0, size().width, size().height);
    paint(g);
}
public void paint(Graphics g) {
    if (md != null) {
        md.mat.unit();
        md.mat.translate(-(md.xmin + md.xmax) / 2,
                         -(md.ymin + md.ymax) / 2,
                         -(md.zmin + md.zmax) / 2);
        md.mat.mult(amat);
        md.mat.scale(xfac, -xfac, 8 * xfac / size().width);
        md.mat.scale(xfac, -xfac, 16 * xfac / size().width);
        md.mat.translate(size().width / 2, size().height / 2, 8);
        md.transformed = false;
        if (backBuffer != null) {
            if (!backSize.equals(size()))
                newBackBuffer();
            backGC.setColor(getBackground());
            backGC.fillRect(0,0,size().width,size().height);
            md.paint(backGC);
            g.drawImage(backBuffer, 0, 0, this);
        } else
            md.paint(g);
        setPainted();
    } else if (message != null) {
        g.drawString("Error in model:", 3, 20);
        g.drawString(message, 10, 40);
    }
}
private synchronized void setPainted() {
    painted = true;
    notifyAll();
}
private synchronized void waitPainted() {
    while (!painted)
    {
        try
        {
            wait();
        }
        catch (InterruptedException e) {
            // Handle the exception
        }
    }
```
```java
class Atom {
    private static Applet applet;
    private static byte[] data;
    private final static int R = 40;
    private final static int hx = 15;
    private final static int hy = 15;
    private final static int bgGrey = 192;
    private final static int nBalls = 16;
    private static int maxr;
    private int Rl;
    private int Gl;
    private int Bl;
    private Image balls[];
    static {
        data = new byte[R * 2 * R * 2];
        int mr = 0;
        for (int Y = 2 * R; --Y >= 0;) {
            int x0 = (int) (Math.sqrt(R * R - (Y - R) * (Y - R)) + 0.5);
            int p = Y * (R * 2) + R - x0;
            for (int X = -x0; X < x0; X++) {
                int x = X + hx;
                int y = Y - R + hy;
                int r = (int) (Math.sqrt(x * x + y * y) + 0.5);
                if (r > mr)
                    mr = r;
                data[p++] = r <= 0 ? 1 : (byte) r;
            }
        }
        maxr = mr;
    }
    static void setApplet(Applet app) {
        applet = app;
    }
    Atom(int Rl, int Gl, int Bl) {
        this.Rl = Rl;
        this.Gl = Gl;
        this.Bl = Bl;
    }
    private final int blend(int fg, int bg, float fgfactor) {
        return (int) (bg + (fg - bg) * fgfactor);
    }
    private void Setup() {
        balls = new Image[nBalls];
        byte red[] = new byte[256];
    }
}
```
```java
red[0] = (byte) bgGrey;
byte green[] = new byte[256];
green[0] = (byte) bgGrey;
byte blue[] = new byte[256];
blue[0] = (byte) bgGrey;
for (int r = 0; r < nBalls; r++) {
    float b = (float) (r+1) / nBalls;
    for (int i = maxr; i >= 1; --i) {
        float d = (float) i / maxr;
        red[i] = (byte) blend(blend(Rl, 255, d), bgGrey, b);
        green[i] = (byte) blend(blend(Gl, 255, d), bgGrey, b);
        blue[i] = (byte) blend(blend(Bl, 255, d), bgGrey, b);
    }
    IndexColorModel model = new IndexColorModel(8, maxr + 1,
                                            red, green, blue, 0);
    balls[r] = applet.createImage(
        new MemoryImageSource(R*2, R*2, model, data, 0, R*2));
}
}
void paint(Graphics gc, int x, int y, int r) {
    Image ba[] = balls;
    if (ba == null) {
        Setup();
        ba = balls;
    }
    Image i = ba[r];
    int size = 10 + r;
    gc.drawImage(i, x - (size >> 1), y - (size >> 1), size, size, applet);
}
```
APPENDIX C

Java Class Hierarchy

OVERVIEW

The Java Class Hierarchy is shown in the next series of pages. Generally, pages are divided by packages, although some packages are too large to fit on a single page. The boxes that are not part of the class library with the centered text serve as keys to the packages represented on that particular page.
JAVA CLASS HIERARCHY

Figure 17

Key

- Class
- Interface
- Extends
- Implements
- * - Duplicated
Object (java.lang)
  BorderLayout
  CardLayout
  CheckBoxGroup
  Color
  Component
    Button
    Canvas
    Checkbox
    Choice
    Container
      Panel
        Applet
      Window
        Dialog
          FileDialog
        Frame
          MenuContainer
  Label
  List
  Scrollbar
  TextComponent
    TextArea
    TextField
Object (java.lang)
  ├── BitSet
  │    └── Cloneable*
  ├── Date
  ├── Dictionary
  │    └── Properties
  │         └── Cloneable*
  │            ├── Hashtable
  │            │     └── Cloneable*
  │            └── Properties
  └── Observable
      ├── Random
      └── StringTokenizer
          └── Enumeration
      └── Vector
          └── Cloneable*
              └── Stack

Observer

java.util
  └── java.lang
LESSON 1

Review questions, page 22

   AdjustmentListener uses adjustmentValueChanged(AdjustmentEvent).

   ItemListener uses itemStateChanged(ItemEvent).

2. The method used is getSource().

3. It offers a way of handling events that are extended from AWT components.

LESSON 2

Review questions, page 38
1. This applet alternates its display between a solid block of red and a solid block of blue. However, it does not work properly because it does not call repaint() to redraw the screen after changing colors.

2. getImage() requires a URL object, not a String object.

3. If the loop is not halted in the applet’s stop() method, the sound will continue to play even after the applet has stopped executing.

LESSON 3

Review questions, page 79
1. java.awt.Container.add()

2. Buttons, Checkboxes, Choices, Lists, and TextFields can all produce action events. So can Menu items, though they are not Components.

3. java.awt.FlowLayout
LESSON 4

Review questions, page 92

1. You must call its `setVisible(true)` method. It may also be necessary to resize it.

2. The default layout manager for all frames is of type `BorderLayout`, which requires that you use the form of the `add()` method that takes a `String` argument describing the position of the component within the container (north, south, center, east, or west). If you omit this `String` argument, the layout manager will ignore the component, and the component will not appear in the frame.

3. A dialog box is a secondary window, has no menu bar, and may be modal.

LESSON 5

Review questions, page 106

1. The parameter to the catch statement requires a name.

2. D E F
   G H I

LESSON 6

Review questions, page 135

1. `InputStream.read()` returns an `int`. If the value is not equal to -1, indicating an error, it may be casted down to a `byte`.

2. When one of the various `DataInputStream` methods encounters an end of file, it throws an exception of type `java.io.EOFException`.


4. The best choice for producing ASCII output is `writeBytes()`, though `writeUTF()` also produces ASCII output if the argument `String` contains only ASCII characters. `writeChars()` writes 16-bit characters for each character in the `String`, ASCII or not.
LESSON 7

Review questions, page 151
1. The method does not compile. The result of `Vector.clone()` is an `Object`, which must be cast down to type `Vector` before it may be assigned to a variable of type `Vector`.

2. The `Vector` method that returns the length of the `Vector` is called `size()`, as opposed to `String.length()` and the length field of arrays!

3. `java.lang.Double`

LESSON 8

Review questions, page 185
1. PropertyChangeListener.

2. The manifest file.

3. The reflection API. More specifically, the `Class.getMethod()` function.

4. Event state objects end in "Event"
   Event listener interfaces end in "Listener"
   Boolean properties use `set<PropertyName>()` for the "setter" method and then either `is<PropertyName>()` or `get<PropertyName>()` for the "getter" method.