# Interactive Visual Programming: Principles and Examples

**Peter Wegner and Joel Silverberg, February 25 1999**

## 1. Introduction

Emerging principles of interactive visual programming are transforming component-based technology from an ad-hoc collection of techniques into a subdiscipline with conceptually deep design principles and semantic models. Visicalc and Excel provided special-purpose interactive visual tools in the 1980s, while Visual Basic and Active X controls have provided general-purpose tools in the 1990s. Java 1.2 systematically coordinates interactive, visual, and algorithmic functionality in a new and interesting way to provide an integrated framework for interactive visual component technology.

Java's interactive visual programming paradigm is a "listening paradigm" closer to human problem solving than algorithm execution. Components have *listening membranes* that are a computational analog of sense organs like our eyes, ears, and skin. Java's "event classes" specify modes of listening and responding to events that transmit stimuli from interface components (event sources) to inner components (event targets) for handling and response. Menu and checkbox event sources in our first example correspond to eyes and ears, while the applet panel (second example) is like the skin.

Interactive programs have characteristic program-structuring and design methods [GHJV] that are coalescing into coherent conceptual models. Java's event-driven component model provides a concrete implementation for interaction machines, which were shown [We2] to be more expressive than Turing machines, and therefore harder to formalize. The Java model is sufficiently clean to serve as a canonical operational semantics for interactive components. The data structure for the Java classes Component and Container (Figure 2 below) play a role similar to stack-based activation record models for block-structure languages.

We present principles of interactive visual programming through "learning (or teaching) by example", by introducing concepts of Java in a top-down way on a need-to-know basis. This article aims to contribute to a better understanding of the fundamental concepts and models of interactive program design by examples that will be useful in the classroom. Our example-driven approach corresponds to learning by listening or reading, allowing educational form to follow function.

Java's software components acquire their interactive visual functionality from class libraries and external events. Programs are constructed by connecting (gluing) instances of classes so they can interact with both each other and external events. Two example applet programs are examined in detail both as an educational case study and to show the structural differences between languages like Pascal and visual interactive programming languages like Java. Working code can be found at *http://conan.ids.net/~joels/applets.*

*ShapesAndColors applet that interactively draws colored shapes on a canvas*

*Scribble applet that supports interactive drawing (scribbling) on the applet's surface*

To provide a framework for examples, we focus on the language and environment mechanisms by which Java's components acquire their visual and interactive functionality. We then explain the code for the ShapesAndColors applet, discuss the effectiveness of "learning by example" as a systematic method for teaching, and finally present the Scribble applet that shows Java's model for low-level mouse events.

## 2. How Components Acquire Visual and Interactive Functionality

Java is an object-oriented language whose objects belong to classes that support inheritance [We1]. We use the term "component" both in an informal sense to mean "heavy" objects that may have interactive and visual functionality, and in a technical sense to mean objects of the Java class Component, which are a particular realization of the informal notion.

*informal definition: a component is an object that may have interactive and/or visual functionality*

*technical definition: a component is an instance of the Java class Component*

Java's components support interaction through an event model that lets components act as event sources that inform listening components of the occurrence of events, and supports visual widgets by a class library of visual components (controls). It integrates interaction provided by its event model with visual function-

ality supplied by library classes and algorithmic functionality supplied by traditional language features.

### 2.1 Is-A, Has-A, and Interacts-With Functionality

*Objects* are instances of classes with copies of instance variables and shared methods inherited from superclasses or declared by their class. Accessibility of declared names may be *public*, *private* (accessible only within the class), *protected* (accessible only in subclasses), or *package* (accessible within package).

Java's components derive their functionality through inheritance, instance variables, and events.

**inheritance**: an object (component) inherits from superclasses of which its class *is-a* subclass

**instance variables**: a component *has-a* collection of instance variables defined in its class

**events**: a component (event source) *interacts with* events and notifies registered listeners

The *is-a* functionality of components is derived through inheritance, the *has-a* functionality through instance variables directly defined in the component's class, and the *interacts-with* functionality by registering listening components with event-source components. Instance variables are references to components that may have visual representations and come with methods for component manipulation derived from the component's class. The Java event model propagates change notification from event sources to listening components which are instances of classes that implement listening interfaces. The event model can be used to handle both external user-initiated event and internal events that make use of the Java event mechanism to notify listeners of the event's occurrence.

Dependence of classes on superclasses is specified by an inheritance hierarchy whose root is the class Object. User applets extend the Java class Applet by the code "**class** UserApplet **extends** Applet". Applet is a fifth-level class that extends the classes Panel, Container, Component, and Object (Figure 1).

*UserApplet extends Applet extends Panel extends Container extends Components extends Object*

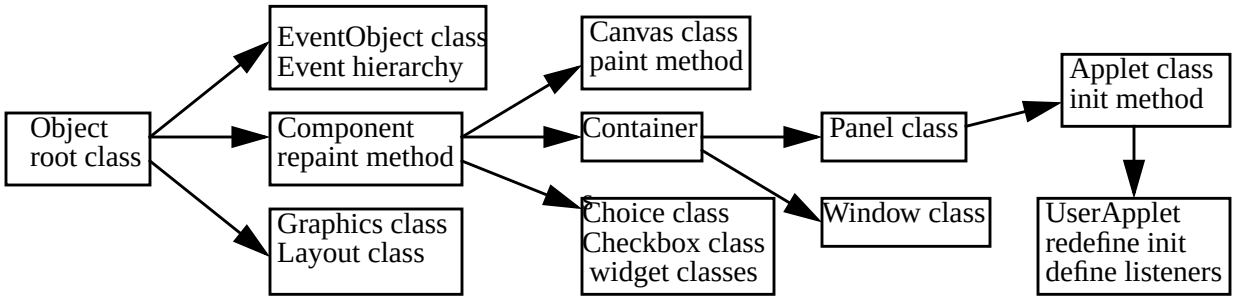



**Figure 1. Fragment of the Java Class-Inheritance Hierarchy**

Each inheritance level of the class Applet adds qualitatively different functionality. Direct subclasses of the root class Object include event model primitives, layout and graphics primitives, and the class Component that is the base class for Java's interactive visual functionality. Subclasses of Component include specific widget classes like Canvas, Choice, and Checkbox classes, and the class Container that can contain multiple components. Containers can be panels whose components have a fixed layout or windows whose components can be moved, resized, and otherwise manipulated. Applets are panels that may contain components whose interactive and visual properties are defined by the applet designer. Instances of the Canvas, Choice, and Checkbox classes are components, while instances of Event and Graphics classes are not.

Each instance of the class Component has a set of required instance variables, including a reference to its parent container, a reference to a graphics object that allows the component to draw itself, and a reference to the list of registered listeners for each listener interface for which that object may be an event source. If the list of registered listeners is empty, then no action need be taken when an event for that listening interface occurs. If there are multiple registered listeners, then occurrence of the event sends (multiplexes) a message to each listening object. Multiplexed notification may be implemented concurrently.

Figure 2 shows that instances of the class Container inherit references to parents, graphics objects, and lists of listeners from Component superclasses and have a required array of references to the components they contain. They inherit the properties of having a parent, the ability to draw themselves, and the ability

to serve as event sources for specified listening interfaces because a Container *is-a* Component, and **have-a** array of references (which is empty if the container has no components). Components cannot contain components, though they are contained in a container.

| **is-a instance variables** | **has-a instance variables** |
|---|---|
| **inherited from component:** | **required container variables:** |
| reference to parent container | array of references |
| reference to graphics object | to container's components |
| reference to list of listeners | empty for empty components |
| for each event listener interface | |

**Figure 2: Instance Variables of Container Classes**

When an instance of a container class, such as an applet, is created, its instance variables include at least the inherited and required instance variables of Figure 2. Creation of an applet essentially entails creating a Panel container class, initializing the reference-to-component array to the visual components of the applet, and initializing the event-source listening lists for the container and its components so that notification of external events will be transmitted to designated listening components.

JavaBeans components [HC], called beans, support a different, though related, class of software components. Beans are components that can be manipulated by a builder that provides a high-level language for customizing and connecting components. They have *properties* with get and set operations, *methods* for operating on properties, and *events* that conform to the Java event model. Beans with a visual interface must be of the class Component, but nonvisual beans that respond to events from other beans need not be subclasses of Components. Not all beans are instances of the class Component and conversely not all Component instances are beans, because beans have restrictions placed on the form of their selectors and on other attributes so they can be manipulated by builders.

In this article we focus on visual interactive functionality provided by subclasses of Component, and look under the hood at how such functionality is implemented. The Java class Component provides a relatively clean model for software components with interactive and visual properties that can serve as a basis for abstraction to general component-based models. There is no generally agreed-on definition of components, just as there is no generally agreed-on definition of objects. But the Java class Component provides a canonical conceptual model for interactive visual components and a simply described canonical implementation (operational semantics) in terms of inheritance, instance variables, and events.

The mechanisms by which containers and components acquire their functionality are very different from those of procedures in languages like Pascal that focus primarily on the noninteractive (algorithmic) transformation of inputs into outputs by statements, control structures, and declarations. The Java environment differs from Pascal not only in being object-oriented but also in providing specific class libraries and event hierarchies and specific mechanisms for incorporating this functionality in software components.

The difference between is-a functionality of inheritance and has-a functionality supplied by instance variables corresponds roughly to that between inherited and acquired characteristics for people. Users do not care whether functionality is inherited or acquired, and it is generally quite easy to take resources that are supplied by instance variables and, through mechanisms like wrappers, make them appear to be innate (inherited) resources. The class library structure determines both implicit resources that components acquire through inheritance and resources that are explicitly acquired through instance variables.

The user applet ShapesAndColors (discussed in section 3) *is-a* Applet which in turn *is-a* panel and container. It *has-a* set of instance variables of the classes Canvas, Choice, Checkbox, and Graphics that determine its visual appearance, and *interacts with* external events through components that implement listening interfaces that let the applet dynamically change the display of its color and shape.

### 2.2 Classes and Interfaces

Classes and interfaces are distinct language constructs. Classes specify both an *interface* of public methods and data and an *implementation* that can be invoked by instances of the class. They may contain

*virtual methods* that are unimplemented in the class and must be defined in subclasses before they can be used. Both implemented and virtual methods may be redefined in subclasses to specialize superclass methods for instances of the subclass. Thus "paint" methods, which are the visual analog of print methods, are often redefined in subclasses that display their visual form on the screen.

Interfaces are "lightweight" classes that require all their methods to be virtual (with no implementation) and impose an obligation on classes to implement all virtual methods of interfaces that they support. Java permits interfaces to extend multiple parent interfaces and classes to implement multiple interfaces. It supports "multiple inheritance" of interfaces by interfaces and classes, but only single inheritance of classes. Interfaces can contain methods and constants but not variables. Formal parameters and instance variables of an interface type have values that are references to components that implement the interface. Java supports interfaces as well as classes as first-class values.

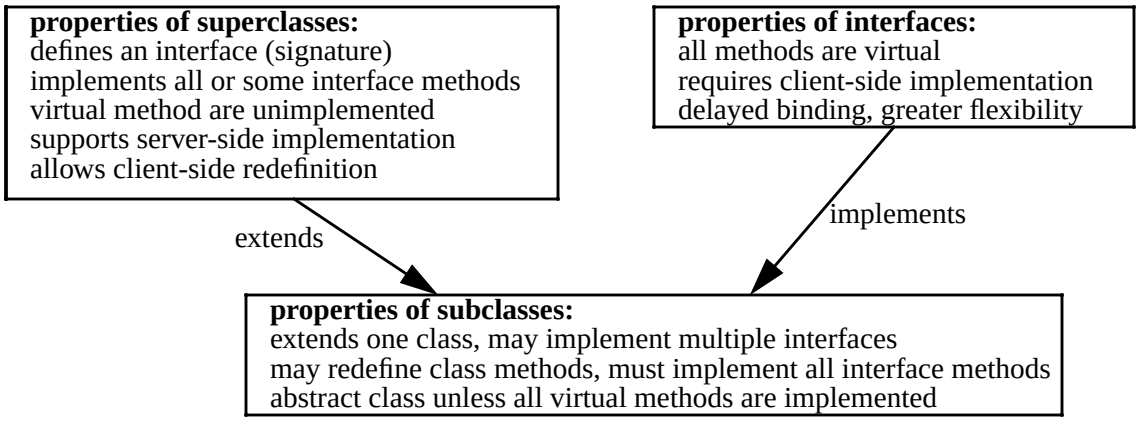



**Figure 3: Properties of Classes and Interfaces**

A class that inherits functionality from a superclass and has multiple listening interfaces for several kinds of dynamically occurring events is illustrated in the ShapesAndColors example (see Figure 5).

*classes extend the functionality of superclasses and implement (provide methods for) interfaces*
*interfaces impose obligations on classes that implement them to provide interface methods*
*classes extend only one superclass but may implement multiple interfaces*

Listening is better implemented by interfaces that allow each listener to respond individually to events than by classes that implement a shared listening method. Listener interfaces require client-side rather than server-side binding of implementation code for interface events. Late client-side binding is clearly more flexible than early server-side binding of the actions associated with interface specifications.

### 2.3 The Java Event Model

Java has many different kinds of events organized into an event hierarchy (Figure 4). Classes that respond to events register their desire to listen to an event with an event source and specify listener components that respond to the occurrence of events. External screen events cause the operating system to generate an event notice that is deposited into an event queue with the event source as a parameter. The AWT processes event notices in the event queue by dispatching (multicasting), to each registered listening component of a designated listening interface and event source, an event object that specifies the nature of the event. Registered listening components implement handlers whose execution is triggered by event objects.

a component *registers events to which it wants to listen with a listener interface of an event source*
*the component's class implements an EventListener interface that handles events*
*external events cause event objects that trigger execution of client-side EventListener implementations*

The control structure for events is a form of exception handling. Registering an event with an event source causes the event source to throw (raise) an exception when the event occurs that will be handled by the registering component. Registration supports a more flexible connection between the source and han-

dler for events than for exceptions, and allows an event to be handled by multiple listeners whereas an exception is handled by just one handler. But the control structure for events and exceptions is fundamentally identical.

The package java.awt.event provides interfaces for handling events generated by interaction with users. The inheritance hierarchy of Java event classes in Figure 4 shows a systematic classification of Java interfaces events and a framework for adding new kinds of events, like audio and multimedia events.

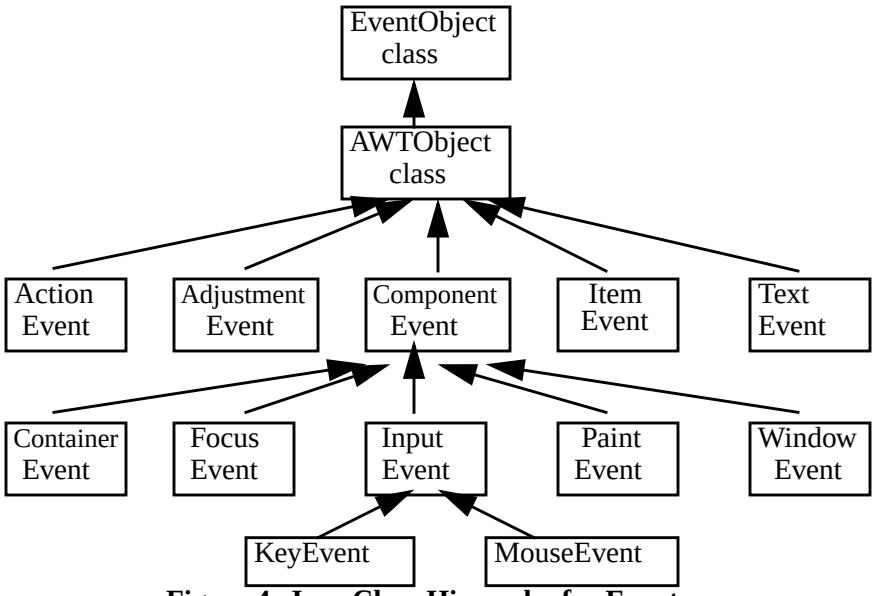



**Figure 4: Java Class Hierarchy for Events**

Objects that respond to events must implement listener interfaces of the associated event class. Some events are generated by low-level user actions such as mouse or keyboard events (KeyEvent, MouseEvent, FocusEvent, ContainerEvent), while others are generated by user interaction with AWT components (ActionEvent, AdjustmentEvent, ItemEvent, TextEvent). Each event class has a listener interface that provides the signature of methods for the event type, which must be implemented by the responding class.

For example, ItemEvents are generated by pull-down menus or by checkboxes when the user clicks on them. An instance of a class that implements the interface ItemEventListener has an ItemEvent interface with a method itemStateChanged that must be implemented by listeners that respond to the event. Mouse events indicate that the user has moved the mouse or pressed one of the mouse buttons. Mouse events have methods that return the number of times the mouse button was clicked and the coordinates of the mouse. They inherit methods to determine the component the mouse was over at the time of the event. Methods are provided to determine if the mouse was pressed, released, clicked, moved, dragged, etc. The interfaces MouseListener and MouseMotionListener provide separate responses to each of these possibilities.

External events of an event class cause event objects of that class to be transmitted from the event source to each listener registered with the event source. ItemEvent objects signal the occurrence of user clicks on menus or checkboxes, ActionEvent objects occur when the user enters data in a textfield or clicks on a button, while MouseEvent objects occur when the user clicks or moves the mouse.

The event types for which instances of an interactive class may listen are a part of the class definition. The class Component has references to lists of listeners for Focus, Key, Mouse, MouseMotion, and Component events. Subclasses of Component can add additional listeners. For example, the class Container adds a listener for Container events and Checkbox adds a listener for Item events.

Interactive execution of sequences of unpredictable screen events is specified and handled very differently from noninteractive execution of inner computation steps:

*noninteractive dynamics: execution of sequences of algorithmically determined computation steps*

*interactive dynamics: execution triggered by external screen events*

The interactive “listening paradigm” is cognitively closer to human behavior than the algorithmic “speaking paradigm”. Listening membranes that respond to a variety of different kinds of events parallel mechanisms of human interactive behavior. Interface stimuli reported to inner handlers that cause actions with visible interface effects are a common model for human and computational interactive behavior. The event model provides an implementation for interaction machines [We2] that have been shown [We3, WG] to provide a unifying model for object-oriented and agent-oriented programming more expressive than Turing machines. The connection of the Java event model to abstract models of interactive computation motivates our interest in event models but is beyond the scope of this paper.

Java’s role in applet programming is that of a modern scripting language. Java provides glue for managing the visual behavior of library-defined widget classes and controlling their behavior. Our applet examples show that Java seamlessly supports scripting for the management and control of components within the framework of general-purpose programming. Implementations of JavaBeans like IBM’s Visual Age explicitly permit the use of Java for scripting when visual methods must be supplemented by programming to realize the desired functionality.

The term “scripting language” was first used in the context of Unix, but has increasingly come to mean a language for specifying and controlling the interactive behavior of visual interfaces. This modern meaning identifies scripting with visual, interactive programming. Java shows that scripting languages need not be specialized languages that sacrifice efficiency for flexibility and that scripting can be included in general-purpose programming languages that have interactive, visual, and algorithmic functionality.

### 2.4 Java Packages: Physical Directory Structure

The Java programming environment consists of the Java language [AG] augmented by class libraries [Fl]. Java’s system resources are implemented by classes grouped into about 20 packages (class libraries) that determine the name space (directory structure) for system-specified resources.

*packages contain classes, interfaces, and other packages as their members*

Language facilities are specified in the package java.lang, which is only a small part of the system resources provided by Java. The remaining packages are specialized class libraries for a variety of particular purposes. The packages particularly relevant to applet programming include:

*java.lang: contains (provides) traditional language facilities*
*java.awt: abstract window tool package that provides visual programming functionality*
*java.awt.applet: includes the class Applet, can be embedded in browsers*
*java.awt.event: event classes and event listener interfaces that provide interactive functionality*

Packages contain information resources (classes and interfaces) accessible to Java programmers by qualified names. Import commands permit imported resources to be referred to directly by unqualified names. For example, the import command “**import** java.awt.applet.*;”, which imports all resources of Java’s awt.applet package, allows its Applet class to be referred to by its unqualified name Applet.

### 2.5 Structure of Applets

Applets execute the two operations **init** and **start** when they are first displayed by the browser and the operations **stop** and **destroy** when they are deleted. Each of these operations may be overridden in classes that extend Applet. We override the initialization method **init** to create the visual interface and register event listeners of the applet with event sources. The code for our example applets has the following parts:

***import** commands for external resources (class-library packages and individual classes)*
***public class** that names the applet and **extends** the class Applet*
*declaration and initialization for variables and visual structures (widgets)*
***init** method that initializes interface and registers event listeners with event sources*
*internal methods with listening interfaces that interpret and respond to interface events*

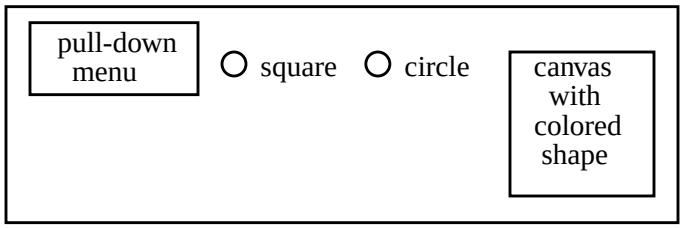
Applet code can be embedded in a Java enabled browser by HTML commands:
<APPLET code = “name.class”, width = display-width, height = display-height> </APPLET>

This code will cause Java-enabled browsers to execute the applet, display its initial visual form in the specified window, and respond to user interface events in a manner that the applet specifies.

### 3. ShapesAndColors Applet

The ShapesAndColors applet, whose complete code is given in appendix 1, is visually represented as a container (panel) with four components, consisting of a pull-down (drop-down) menu to choose among colors, two checkboxes (radio buttons) for choosing among shapes, and a canvas on which colored squares and circles are drawn.

The user of the applet can click on the menu to change the color and on square or circle checkboxes to change the shape appearing on the canvas. Its menu, square, and circle components are event sources for ItemEvents with the canvas component as their target. Thus the canvas component will register with menu, square, and circle components so that it can respond to events that change the state of these event sources.




**this applet panel contains four widgets**

**pull-down menu controls the color**
**two checkboxes control the shape**
**canvas is the target of all changes**
**interactively changes its color and shape**
**in response to menu and checkbox events**

**Figure 5: The Panel of the ShapesAndColors Applet**

Clicking on the menu exposes a list of color names that can be selected to change the color. Clicking on the menu or checkboxes generates ItemEvent objects that are sent to the canvas. The canvas is an instance of the inner class DrawOn that implements listener interfaces for menu and button events to which the canvas wishes to listen and has a paint method for drawing (painting) colored shapes on the canvas.

The Applet data structure has the container instance variables of Figure 2. Its array of references to components will have four elements, corresponding to the four components of Figure 5. Since the applet is a subclass of Panel, it will be an event source for MouseEvents and MouseMotionEvents (Figure 6).

| **inherited from Component:** | **required container variables:** |
|---|---|
| reference to parent container | ref to menu component |
| graphics object reference | ref to circle component |
| ref to MouseListener list | ref to square component |
| ref to MouseMotionListener list | ref to canvas component |
| ref to other listener interface lists | |

**Figure 6: Instance Variables of ShapesAndColors Panel (Container)**

Though an applet panel can listen for MouseEvents and MouseMotionEvents, the event sources in this example are the menu, square, and circle components rather than the applet itself, so there will never be registered listeners for mouse events. Menu, square and circle components are event sources for ItemEvents with nonempty lists of registered listeners whose target is the canvas.

#### 3.1 Applet Structure and Imported Resources

The ShapesAndColors applet has the following structure:

*import commands and class declaration that extends Applet*
*instance variables for menu and checkbox event sources and the canvas event listener*
*init method that places widgets on the panel and registers the canvas as a listener*
*DrawOn class that extends Canvas and implements ItemListener to handle events*

It imports the packages java.awt, java.applet, and java.event and is introduced by a class declaration that extends the class Applet.

```
import java.awt.*; // set of all resources in the package java.awt
import java.applet.*; // set of all resources in the package java.applet
```

**import** java.awt.event.*; // resources in the package java.awt.event, used for event handlers
**public class** ShapesAndColors extends Applet

Import commands do not actually import resources. They allow direct (abbreviated) naming of classes specified in imported packages. For example, **import** java.applet.*; allows the class whose full name is java.applet.Applet to be referred to by its abbreviated name Applet.

### 3.2 Declaring the Applet's Instance Variables

Instance variables are declared by specifying the variable's class and creating an initialized instance.

*Class variable; // declare a variable of the given class, usually a reference to a component*
*variable = new Class(parameter-list); // create and initialize variable to an instance of the given class*

Declaration and instance creation can be specified by a single statement:

*Class variable = new Class(parameter-list); // declare variable, create instance, assign to variable*

The named variable is first created with a null reference, a new instance of the class is then created and initialized by the parameters, and a reference to the newly created instance is assigned to the variable. The parameter list may be empty, indicating that class parameters are initialized by default rather than by user-supplied values. Local (hidden) instance variable declarations are preceded by the keyword **private**. Applets may declare variables for classes defined in accessible (public) packages.

Applets are panels (containers) whose instance variables include both components that have a visual appearance when added to the panel and auxiliary objects that play a role behind the scenes. The instance variables for the applet specify menu, checkbox, and canvas components to be displayed on the screen. Variables that represent components with a visual appearance are usually instances of public classes.

The first declaration creates an instance variable of the class Choice (that extends Component), which specifies pull-down menus. The variable is called color because it will specify choice of a range of colors.

**private** Choice color = new Choice(); // create a pull-down (drop-down) menu

The instance variable "color" is a component whose applicable operations are determined by the class Choice, which is a subclass of Component (Figure 1). The class of other declared instance variables likewise determines their operations and allows the applet to exhibit functionality of associated classes.

The declared variable shapes of the class CheckboxGroup (that extends Object) defines a group of checkboxes (radio buttons).

**private** CheckboxGroup shapes = new CheckboxGroup(); // create a coordinator for checkboxes

The next two declarations create two variables "square" and "circle" of the class Checkbox (which extends Component) and initialize their properties with parameters. The three parameters of Checkbox specify the printname, whether initially checked, and the CheckboxGroup (shape) to which the checkbox is assigned. The Circle checkbox is initialized to true and will therefore be initially displayed.

**private** Checkbox square = new Checkbox("Square", false, shapes);
**private** Checkbox circle = new Checkbox("Circle", true, shapes);

The next two declarations specify an array of the type Color (which extends Object) and an array of the type String (for naming colors). The class Color has values that are physical colors, represented by (red, green, blue) triples. The array theColor of the class Color is initialized to the three system-defined colors red, green, and blue, and the associated array colorName is initialized to the color names "Red", "Green", "Blue". The color names will later be inserted in the pull-down menu and will be used to select corresponding colors in the Color array.

**private** Color [] theColor = {Color.red, Color.green, Color.blue};
**private** String[] colorName = {"Red", "Green", "Blue"};

The final instance variable declaration declares an instance of the user-defined class DrawOn, defined in the body of the ShapesAndColors applet, that extends the class Canvas, which is itself defined as an extension of the class Component.

**private** DrawOn canvas = new DrawOn(); // create canvas as instance of DrawOn that extends Canvas

The instance variables of the ShapesAndColors applets are more complex than types like integer, Bool-

ean, character, array, and record that occur in traditional programming languages, since they include visual as well as logical properties. Moreover, they may be defined in inherited class libraries in remote regions of the inheritance hierarchy. The classes of the seven instance variables of the ShapesAndColors applet include one primitive language type, two classes that extend Object, three classes that extend component, and one user-defined type that extends Canvas (that is an extension of Component).

### 3.3 Initializating the Applet's Interactive Visual Interface

Having defined the instance variables, we are ready to define the **init** method of the applet, whose execution creates its initial screen representation. The **init** method is a public procedure with no return value.

**public void init**() {

The next four lines add to the Applet panel the color menu, the square and circle radio buttons, and the canvas that will contain the colored shape. The default layout of this panel causes items to be added as a sequence, with wraparound to the next line when there is no more space.

```
add(color);  // add the pull-down menu called color to the applet panel
add(square);  // add the checkbox square to the applet panel
add(circle);  // add the checkbox circle to the applet panel
add(canvas);  // add the canvas to the applet panel
```

The following code adds the color names to the pull-down menu called color. It could have appeared prior to the code for adding the menu to the panel. Widgets can be initialized either before or after they are added to a display.

```
for (int i=0; i < colorName.length; i++)
 color.addItem(colorName[i]);  // addItem is a method for adding menu items to Choice menus
```

The color variable is initialized so that the menu displays the first color (red) as its initial color, and the canvas size is initialized to 150 by 150 pixels. "select" is a method of the Choice class, while "setSize" is a method that the class Canvas inherits from the class component. These initializations can likewise be done before or after items are displayed.

```
color.select(0);  // initially display the zeroth string (red) in the pull-down menu
canvas.setSize(150,150); // set size of canvas to 150 by 150 pixels
```

When users click on checkboxes or pull-down menus, the mouseUp action generates ItemEvents. Components that listen to ItemEvents must implement an ItemListener interface that contains a method itemStateChanged with an ItemEvent parameter.

The final code in **init** registers the canvas as a target of item events generated by the event-source components color, square, and circle when the user clicks on them.

```
color.addItemListener(canvas);
square.addItemListener(canvas);
circle.addItemListener(canvas);
```

The command."s.addItemListener(canvas)" registers canvas as a component to be informed of ItemEvents generated by the user with the component "s". This commits "canvas" to implement the ItemListener by providing a method itemStateChanged that defines how canvas will respond to ItemEvents.

The instance variable declarations specify the collection of components (beans) of the applet in terms of Java classes, while the init method places the components on the panel (beans in the beanbox), customizes the components (beans), and establishes connections between event sources and listeners. Programmed widgets are more flexible than widgets specified in a higher-level language like JavaBeans, and allow looking under the hood to examine implementation mechanisms.

### 3.4 DrawOn Class that Extends Canvas and Listens to Menu and Checkbox Events

The remaining code of the Applet implements a class DrawOn that extends the AWT-defined class Canvas, which is a subclass of Component and implements an ItemListener interface that performs actions when DrawOn instances are notified that an ItemEvent has occurred for color, circle, or square.

**class** DrawOn **extends** Canvas **implements** ItemListener

Class definitions extend a single parent class and may additionally implement several interfaces whose virtual methods must be implemented in the defined class. The class DrawOn extends Canvas to inherit the painting methods of Canvas and its superclass Component, and implements an ItemListener interface to specify operations that handle item events. The ItemListener interface requires DrawOn to implement the method itemStateChanged that takes an ItemEvent object as a parameter. In this case the listening action calls the method repaint, defined in Component, to repaint the canvas when designated events occur.

```
public void itemStateChanged(ItemEvent e)
  {  if (e.getStateChange() == ItemEvent.SELECTED)
       repaint();  // repaint canvas, the target of the addItemListener that registered the item event
```

The method itemStateChanged(ItemEvent e) calls the inherited method repaint() for events of the type SELECTED (a MouseUp event for pull-down menus or checkboxes). Without this condition the state would be changed also when the menu or checkbox was deselected by mouseDown events. The test ensures that repainting will occur only once, on mouseUp events.

The repaint() method invokes, behind the scenes, an operating system call to an update(Graphics g) method that redraws the background of the given component followed by a paint(Graphics g) method that DrawOn redefines to perform the drawing of colored squares and circles for the ShapesAndColors applet. The paint(Graphics g) method of Component is redefined below and is passed the graphics object of the canvas as a parameter so that operations like g.setColor and g.fillOval perform paint operations on the canvas. Paint is called indirectly as a result of calling repaint when an ItemEvent occurs.

```
public void paint(Graphics g) {
 g.setColor(theColor[color.getSelectedIndex()]);  // setColor is a method of the Graphics class
if (circle.getState())  // getState is a method of the Checkbox class
  g.fillOval(20,20,100,100);  // fillOval is a method of the Graphics class
else  // square.getState() is true
  g.fillRect(20,20,100,100);
```

Whenever the canvas is painted, this method sets the color of its graphics object to the currently selected color and draws the shape indicated by the checkboxes. The getSelectedIndex method of color is used to index the array of colors, providing the correct color as the parameter for the setColor method of the canvas' Graphics object. It then tests whether the circle checkbox is on (true), using the getState method of the Checkbox class. It draws a filled-in oval if "circle" is checked and a filled-in rectangle otherwise. The fillOval and fillRect commands have four parameters: the first two specify the coordinates of the northwest corner and the last two specify the width and height of the component being drawn.

Understanding this paint method requires knowledge of the environment, including knowledge of methods in the Graphics class that extend Object and of the Choice and Checkbox classes that extend Component. The different mechanisms by which these methods are made available (through instance variable declarations, parameters, and inheritance) are nontrivial. However, the environment knowledge needed to understand "paint" is limited. Understanding of the fact that the Graphics class defined at the level of Object has setColor and fillOval methods while the Choice class defined at the level of component has a setSelectedIndex method and Checkbox has a getState method allows us to understand the structural dependence of "paint" on its environment.

This paint method may also be invoked by calls to repaint() generated by the system in response to the user moving, resizing, reshaping, covering, or uncovering the window containing the Applet. It is not restricted to responding to events reported to the canvas in its role as ItemListener.

## 4. Learning (Teaching) by Example

Analysis of the method of presentation and difficulty of concepts of the ShapesAndColors applet illustrate the process of "learning by example". Instead of starting with the language, Java is introduced as a collection of physical packages (class libraries) whose classes may be instantiated by components that make use of externally defined functionality through is-a, has-a, and interacts-with relations. The name

space of physical packages is contrasted with logical relations among classes specified by inheritance, instance variables, and events. The class Applet is presented as a container (panel) that may have components and interact with (listen to) external events.

The Java environment allows class declaration and instantiation to make use of inherited and public resources, while the event model provides a dynamic environment for systematic interaction with a variety of event classes organized in an event hierarchy. The listening metaphor relates the computational mechanisms of interaction to human mechanisms of perception and problem solving. Item events for menus and checkboxes, action events for text entry and buttons, and mouse events for mouse movements are computational modes of listening. The overall structure of applet code as a class that extends Applet, which in turn extends the class Panel, illustrates how applets are integrated into the Java environment. Embedding of applets in HTML browsers is briefly described to indicate the role of browsers.

Though is-a and has-a functionality is acquired in distinct ways, how functionality is delivered to users does not depend on its mode of acquisition. The user cannot tell how an applet's graphics functionality was acquired, and knows simply that the applet and its canvas act as though it can draw and paint in response to menu and checkbox events.

The high-level discussion of relations among Java classes and components provides a framework for specific applet examples. The visual appearance of the applet and its interactive behavior are presented, showing the menu, checkboxes, and canvas on a screen and the impact of clicking on the menu and checkboxes. The intuitive immediacy of the example, supplemented by a high-level understanding of how applets can utilize inherited resources, prepares the reader for a detailed discussion of applet code.

The import commands and class declaration are easily understood in terms of the earlier discussion of packages and inheritance. The declarations of color and shape as instances of the AWT-defined classes Choice and Checkboxgroup includes a brief discussion of the important topic of creation and initialization of declared variables. Declaring square and circle to be instnaces of Checkbox requires parameters to initialize the properties of checkboxes. The declaration of the Color array and String array needs a short discussion of how colors are represented. Declaration of "canvas" as an instance of DrawOn before the class itself has been defined is perhaps the hardest declaration to understand and is further examined below (Figure 7). The declarations illustrate ideas that are quite complex to present in the abstract, but are simpler in the context of specific examples where only a reading knowledge of the code is required.

The **init** method starts by adding the menu, circle and square checkboxes, and the canvas to the applet panel. It initializes the menu to the color names Red, Green, Blue, sets the initial color to red and sets the canvas size. The final three lines of code of **init** registers the canvas as a target of three ItemListener events associated with clicking on menu and checkbox items. Compared with the declarations, the code of the init method is relatively easy to understand. But the notions of add commands that add widgets to a panel, initialization of visual properties of widgets, and registering of events are unusual operations that need to be systematically described in courses on interactive computing.

The class DrawOn (Figure 7) extends Canvas and implements ItemListener. DrawOn inherits methods

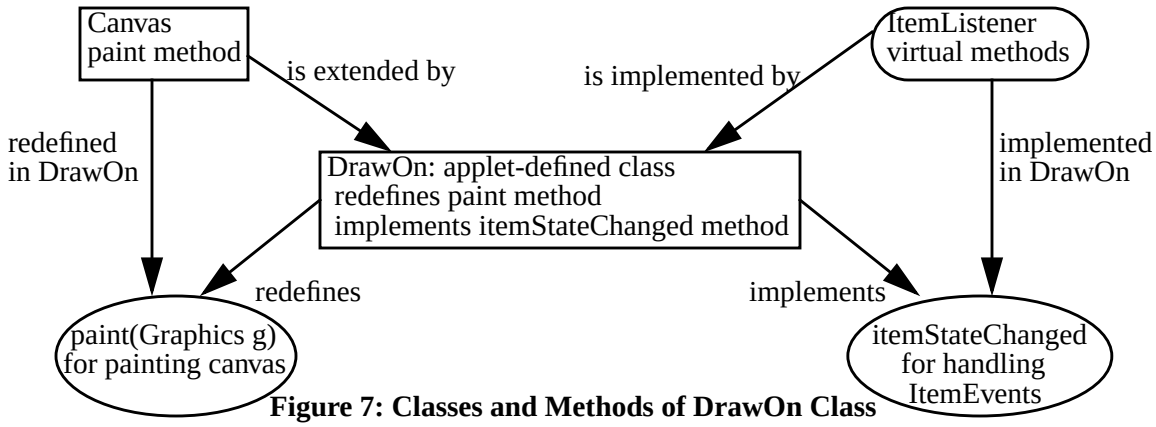



**Figure 7: Classes and Methods of DrawOn Class**

from both the class Canvas and the interface ItemListener and exhibits a form of multiple inheritance. DrawOn has the obligation to implement the methods of ItemListener but there is only the one method ItemStateChanged that needs to be implemented. DrawOn also redefines the method paint inherited from Canvas but in fact defined in the superclass Component. There is much going on behind the scenes when DrawOn responds to an event by executing the itemStateChanged method and calling the inherited repaint method, which in turn calls the redefined paint method, as shown in Figure 7.

Understanding how DrawOn draws on information from the Canvas class, ItemListener interface, Graphics class, Component class, and Object class in realizing its functionality requires smarts about what goes on behind the scenes that are very different from those required to program and understand complex algorithms. However, once the environment structure for this particular example is understood, the mental tools for other applets have largely been mastered. The forms of dependence of DrawOn on its static environment through Canvas and dynamic environment through ItemListener parallel those of human interaction with a predictable physical environment and unpredictable temporal events.

## 5. Scribble Applet

The Scribble applet allows users to scribble directly onto the applet panel by pressing and dragging the mouse. It is simpler than the ShapesAndColors applet in not requiring system-defined widgets for interaction, but has to deal with the low-level complexity of pressing, dragging, and moving the mouse. The applet panel is the computational analog of the human skin, while widgets are a computational analog of human sense organs. Scribble's only widget is a clear button that clears the panel of scribbled drawings.

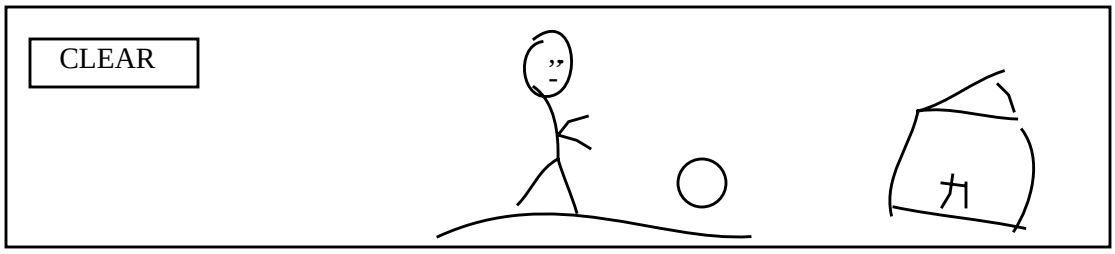



**Figure 8: Scribble Applet with a Clear Button**

The Applet data structure has the container instance variables of Figure 2, with a component array with a single visual element (the clear button). The applet itself is an event source that listens to both mouse clicks and mouse motion. Its MouseListener and MouseMotionListener lists will be nonempty. In fact, the applet is both the source and the target for mouse events, so that external events of mouse clicking and motion will cause the applet to send a message to itself to perform scribbling actions on the applet's screen.

The structure of the scribble applet's code is like that of the ShapesAndColors applet.

import commands, Scribble class declaration that extends Applet, instance variables
init method that registers two mouse listeners and a button listener with the applet
inner classes that implement the mouse and button listeners

### 5.1 Imported Packages and Instance Variables

The import commands, class declarations, and instance variables are similar to but simpler than those of the ShapesAndColors applet.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;
public class Scribble extends Applet {
int lastX;  // initial x coordinate for scribbling
int lastY;  // initial y coordinate for scribbling
```

### 5.2 The Init Method

The Scribble class redefines the **init** method of the class Applet, placing a clear button on the applet panel, setting the panel's background color, registering a button listener, and registering listeners for pressing and dragging the mouse. The declaration for the button b could have appeared outside the **init** method, like the widget declarations of the ShapesAndColors applet. Since the button is referred to only within the init method (as the source for ClearScreenHandler), it is appropriate to declare the button b within **init**.

```
public void init {
Button b = new Button("Clear");  // create a button variable labelled Clear
this.add(b);  // display the button as a widget on the panel
this.setBackground(Color.yellow);  // set this applet's background color to yellow
b.addActionListener(new ClearScreenHandler());  // register ClearScreenHandler as a listener for b
this.addMouseListener(new StartLineHandler() );  // register (add) MouseListener event for applet
this.addMouseMotionListener(new ContinueLineHandler()); // register MouseMotionListener event
}  // end init
```

The last three commands of **init** register three listeners and create three listening components for button and mouse events as a side-effect of registration. The classes for each listener are specified as inner classes of Scribble later in the code. Creation of a new listener of a subsequently defined class at the time of registration is a neat programming technique used both in this and the prvious example.

Components that listen for button and mouse events are created when the events are registered. The occurrence of "**new** ClearScreenHandler" as a parameter of "b.addActionListener" both creates an instance of the class ClearScreenHandler (defined as an inner subclass) and registers it with the button b.

Clicking a button causes an ActionEvent. The ActionListener for the button-clicking event is implemented and handled by the applet-defined class ClearScreenHandler, which is registered with the button b. The command b.addActionListener(new ClearScreenHandler()); registers an Action listener with the button b as its source and instantiates the listener as an instance of the class ClearScreenHandler(), which is defined in later code as an inner applet class.

The source for MouseListener events is the applet panel itself, denoted by "this". MouseListener events are handled by the StartLineHandler class and cause a mousePressed method to be executed when the mouse is pressed anywhere on the applet, while MouseMotionListener events are handled by the ContinueLineHandler class and cause a mouseDragged method to be executed when the mouse is dragged.

### 5.3 Adapters: Auxiliary Classes (Wrappers) for MouseListener Interfaces

Adapters are a general mechanism for enhancing interoperability by interposing modules that perform adaptation of interfaces and enhancement of functionality between modules that perform an actual task. Event adapters interpose adaptation modules between an event source listener list and its target. The Mou-

seAdapter is an auxiliary abstract class interposed between event sources for MouseEvents and their targets to reduce the work of classes that respond to MouseEvents by providing dummy methods for all six of the MouseListener methods in the MouseAdapter. The class MouseAdapter is an example of a “wrapper” that “wraps” the MouseListener interface in an implemented class and allows its listening functionality to be used more easily. The concept of wrappers that enhance portability by adapting interfaces and of adapters that enhance interoperability is nicely illustrated by this example. The MouseAdapter is a system class that is not a part of the applet code, but is included to show how adapters work.

```
public abstract class MouseAdapter extends Object implements MouseListener {
public void mouseClicked (MouseEvent e);
public void mouseEntered (MouseEvent e);
public void mouseExited (MouseEvent e);
public void mousePressed (MouseEvent e);
public void mouseReleased (MouseEvent e); }
```

### 5.4 Handler Classes for Mouse and Button Events

Scribble has two inner classes for listening to mouse events and a third class for listening to action events for the Clear button. Instances of these classes are defined in earlier code at the time listeners are registered.

The class StartLineHandler extends the system-defined MouseAdapter class, which implements stubs for all methods of a MouseListener, and then redefines the mousePressed method, which is the only one needed for scribbling. The action of the mousePressed method is simply to initialize the starting point for scribbling.

```
class StartLineHandler extends MouseAdapter{  // extend adapter class with dummy interface methods
public void mousePressed(MouseEvent e) {
lastX = e.getX();  // x coordinate for start of a scribble
lastY = e.getY();  } } // y coordinate for start of a scribble
```

The class ContinueLineHandler extends a MouseMotionAdapter by redefining its mouseDragged method. It gets the graphics object of the applet panel so that its graphics operations can be used, sets the color of the draw program, draws a line from the start point to the point of the current MouseEvent, and then reinitializes the starting point for the next MouseEvent. Dragging causes about 10 MouseEvents per second, so that scribbles are actually sequences of short lines having the appearance of a curve.

```
class ContinueLineHandler extends MouseMotionAdapter {
public void mouseDragged(MouseEvent e) {
Graphics g = getGraphics();  // returns a reference to the graphics object of the applet
int x = e.getX();  // coordinates of current mouseEvent, supplied by parameter
int y = e.getY();
g.setColor(Color.red);  // uses the setColor method of the graphics object
g.drawLine(lastX,lastY,x,y);  // draw line from last to current coordinates
lastX = x;  // reinitialize coordinates for next drawLine execution
lastY = y;} }
```

The mouseDragged method gets the graphics object of the applet by assigning to g a reference to the graphics object, while the paint method in the ShapesAndColors applet gets the graphics object for the canvas through a parameter. These methods are linguistically different though equivalent from the viewpoint of the methods that use the graphics object. Both methods have the effect that g is an alias for the graphics object that refers to a shared copy implicitly created when the canvas and applet are created.

The ClearScreenHandler class implements an ActionListener - a higher-level event listener triggered by button clicks. It gets the applet’s graphics object, sets the color to the background color for the applet panel, and fills in the complete panel area with the background color.

```
 class ClearScreenHandler implements ActionListener
{ public void actionPerformed(ActionEvent e)
{ Graphics g = getGraphics();  // returns a reference the applet's graphics object
g.setColor(getBackground());  // uses operations of the graphics objects
g.fillRect(0,0,getSize().width, getSize().height);
} } // end Scribble
```

The Scribble applet has the same structure as the ShapesAndColors applet, with some initial import commands and declarations, an init method, and inner classes that implement listeners. It introduces new MouseEvent classes and associated listeners but for the most part serves to reinforce notions of visual and interactive applet programming presented in the first example.

## 6. Conclusions

Our example-driven approach allows new concepts of interactive programming to be presented in a direct and relatively simple manner that corresponds to learning by listening and reading. We believe that programming languages should be assimilated by reading before learning to write, just as natural languages are assimilated by listening before learning to speak. Good programming can be better achieved by learning good programming practice through reading than by learning to program from a set of rules.

This presentation assumes that readers have a basic knowledge of object-oriented programming comparable to an introductory course. But the example-driven approach could easily be extended to more elementary material by adding more basic examples of expressions, statements, control structures, procedures, types, classes, and objects. By focusing on examples rather than on rules of construction of programs, the amount of material that needs to be mastered is dramatically reduced. The effect is more dramatic when applied to complex systems like Java than to simpler systems like Pascal. However, environments that rely on class libraries to support visual and interactive programming will be the norm rather than the exception in the future, and programming by example rather than by language rules may well become the norm for learning and teaching.

## 7. References


[AG] Ken Arnold and James Gosling, *The Java Programming Language*, Addison Wesley, 1996.
[Fl] David Flanagan, *Java in a Nutshell*, 2nd Edition, O'Reilly, 1997.
[GHJV] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[Ha] Graham Hamilton Ed., JavaBeans, Sun Microsystems, java.sun.com/beans
[We1] Dimensions of Object-Based Languages Design, OOPSLA 1987.
[We2] Peter Wegner, Why Interaction is More Powerful Than Algorithms, *CACM*, May 1997.
[We3] Peter Wegner, Interactive Foundations of Computing, *Theoretical Computer Science*, Feb. 1998
[WG] Peter Wegner and Dina Goldin, Interaction, Computability, and Church's Thesis, Brown Technical Report, www.cs.brown.edu/people/pw


**Appendix 1:** Code for ShapesAndColors Applet

```
import java.awt.*;                  // needed for widgets of AWT
import java.applet.*;          // needed for Applets
import java.awt.event.*;       // needed for event handlers

public class ShapesAndColors extends Applet   {
{      // create an empty drop-down menu
       private Choice color = new Choice();
       // Create square and circle radio-buttons with circle selected initially.
```

```
        // Selecting one checkbox within the checkboxgroup will turn the others off.
        private CheckboxGroup shapes = new CheckboxGroup();
        private Checkbox square = new Checkbox("Square", false, shapes);
        private Checkbox circle = new Checkbox("Circle", true, shapes);
        // Initialize an array of strings ("labels" for drop-down menu)
        // and an array of abstract "colors" for setting the drawing color.
        private String[] colorName = {"Red", "Green", "Blue"};
        private Color [] theColor = {Color.red, Color.green, Color.blue};
        // create an instance of DrawOn (canvas to draw upon, with built in eventlistener)
        // class DrawOn is defined later in the code.
        private DrawOn canvas = new DrawOn();

        public void init() {
              // add the pull-down menu color, the Checkboxes square and circle
              // and the DrawOn instance, canvas to the applet container
              this.add(color);
              this.add(square);
              this.add(circle);
              this.add(canvas);
              // Complete the intialization of the drop-down menu, color
              for (int i=0; i < colorName.length; i++)
                color.addItem(colorName[i]);
              color.select(0);  // set the first item to appear as the intial selection
              // Complete intialization the DrawOn instance, canvas
              canvas.setSize(150,150);

              // register canvas as the ItemListener for any ItemEvents generated
              // by the drop-down menu or by either checkboxe
              color.addItemListener(canvas);
              square.addItemListener(canvas);
              circle.addItemListener(canvas);
          }  // end init

// extend Canvas and implement ItemListener
// to provide for both drawing and event handling capabilities
class DrawOn extends Canvas implements ItemListener {

  public void itemStateChanged(ItemEvent e)
  {  if (e.getStateChange() == ItemEvent.SELECTED)
        repaint();  // calls a redefined (below) repaint
                          // to draw the correct object and color
  }
// Redefine DrawOn's paint method (inherited from Component)
// so that whenever a DrawOn instance is asked to draw itself, it will do so
// according to the color indicated by the drop-down menu and the
// shape indicated by the checkbox
  public void paint(Graphics g) {

    // find the index of the selected string in the drop-down menu
    // extract that index from the array of abstract colors.
```

```
   // use the color extracted to set the color of the Graphics object.
   // Anything drawn on the graphics object will be in that color until
   // the color of the graphics object is reset
      g.setColor(theColor[color.getSelectedIndex()]);
      if (circle.getState())
            g.fillOval(20,20,100,100);
      else  // square.getState() is true
            g.fillRect(20,20,100,100);
   }    }    }
```

**Appendix 2:** Code for the Scribble Applet

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

public class Scribble extends Applet {
  // declare instance variables lastX and lastY
      int lastX;
      int lastY;

      // Scribble contains only one method: init()
      public void init() {

            this.setBackground(Color.yellow);  // change applet's background color

            // Create a new button and add it to the applet container
            Button clearBtn = new Button("Clear");
            this.add(clearBtn);

            // Create a listener for button actions and register it with the button
            clearBtn.addActionListener(new ClearScreenHandler());

            // Create listeners for mouse events and register them with the applet
            this.addMouseListener(new StartLineHandler() );
            this.addMouseMotionListener(new ContinueLineHandler());

    } // end init
      // define the three eventListener Classes
      // The applet (container) generates events for the first two MouseEvent
      // event handlers, but the button generates events for the ActionEvent handler

         class StartLineHandler extends MouseAdapter{
            public void mousePressed(MouseEvent e) {
                  lastX = e.getX();
                  lastY = e.getY();
                  }     }
         class ContinueLineHandler extends MouseMotionAdapter {
            public void mouseDragged(MouseEvent e) {
                  Graphics g = getGraphics();
                  int x = e.getX();
```

```
            int y = e.getY();
            g.setColor(Color.red);
            g.drawLine(lastX,lastY,x,y);
            lastX = x;
            lastY = y;
          }    }
class ClearScreenHandler implements ActionListener
{ public void actionPerformed(ActionEvent e)
    { Graphics g = getGraphics();
      g.setColor(getBackground());
      g.fillRect(0,0,getSize().width, getSize().height);
    }      }       }
```