Providing Information (Accessors)

C++ allows you to declare a method as `const`. As a result the compiler warns you if statements in the method change the object’s state.

Java has no such facility, and so it’s up to you to make sure that accessor methods don’t change the state of their objects.

Unfortunately, in Java it is very easy for this to occur. Since object variables are references when your accessor method returns a reference to an object, the caller may be able to modify your object.

```
// Every GoodKid is constructed to stay at a certain
// position relative to the static final field HOME.

public class GoodKid extends Component
{
    public static final Point HOME = new Point(100, 100);
    public static final Point SIZE = new Point(25, 25);

    private Point myLocation;

    public GoodKid( int hPos, int vPos )
    {
        setSize( SIZE.x, SIZE.y );
        myLocation = new Point( hPos, vPos );
    }

    public void paint( Graphics g )
    {
        setLocation( HOME.x + myLocation.x, HOME.y + myLocation.y );
        g.fillOval( 0, 0, SIZE.x - 1, SIZE.y - 1 );
    }
```
// Makes the GoodKids follow the mouse by
// redefining the "constant" GoodKid.HOME
//

public class PiedPiper extends Applet
        implements MouseMotionListener
{

    public void init()
    {
        setLayout( null );
        add( new GoodKid( -20, -20 ));
        add( new GoodKid( -20, 20 ));
        add( new GoodKid( 20, -20 ));
        add( new GoodKid( 20, 20 ));

        addMouseMotionListener( this );
    }

    public void mouseMoved(MouseEvent evt)
    {
        Point bad = GoodKid.HOME;
        bad.setLocation( evt.getX(), evt.getY() );
        repaint();
    }
}
The *GoodKid* class looks like the epitome of well-behaved encapsulation. The field *myLocation* is *private* and it’s set only in the constructor.

The *PiedPiper* class, shows that things are not as simple as they seem. It does not have access to the private data, but it does have access to the *public static* *HOME*.

By using the method *setLocation()* inherited from the *Applet* class, the *Piper* is able to lead the children astray.

*HOME* is final, therefore its value cannot be changed. *HOME* does not contain a *Point*, it merely refers to one. As long as it refers to the same *Point*, its value has not changed.

This does not rule out the possibility that some of the attributes of *Point* might be changed. As a result, of the *setLocation()* now *HOME* has a new location.

You might feel that you could avoid this problem by making *HOME private* and using an accessor to return its value:

```java
private static final Point HOME = new Point(10,10);
public static Point getHome() { return HOME; }
```

However, if done as shown the problem is not alleviated at all.

To avoid this potential time-bomb, you *must* make sure that accessor method return only values, *never* references.

Make sure that all accesses receive a *copy* of the object or, better yet, don’t return access to individual fields at all.

```java
p. s. f. Point HOME = getHome();
p. s. Point getHome() {return new Point (10,10)}
```
An Inheritance Checklist

Use final classes when . . .

- Your class should not be extended.

Use final methods when . . .

- You want to model a strict subtype relationship between two classes. Your saying that the method represents an invariant implementation for all subclasses of this class.

- A method performs an operation in some way that must not change when new classes are created.

Use abstract methods when . . .

- All subclass objects must perform an operation, but the superclass can’t provide a default implementation.

- You intend to write code in terms of superclass objects but, in fact, will use only subclass objects.

- You want to force a subclass to implement a certain behavior, such as initialization.

Use overridden methods when . . .

- A superclass can define some reasonable default action for a method.

- More than one subclass can make use of the default behavior. If only one subclass uses it, then it should be an abstract method of the superclass.
• Subclass objects will perform the same operation in different ways, yet make use of some common behavior to carry out that operation.

If more than one subclass can use the default operation unchanged, use an overridden method.

If all subclasses need to add to or extend the method, place the common code in a protected helper method in the superclass, and make the overridden method an abstract method. If not, authors of subclasses are likely to forget that they must override the method.

**Use purely abstract classes when . . .**

• You can define a common data member in the superclass, even though you can define no methods.

• You’re designing a hierarchy that’s expected to model closely related classes.

**Use interfaces when . . .**

• You’re designing a set of classes that share a common behavior yet aren’t otherwise related.

• There’s no common implementation you want to inherit, but you want to impose a mandatory interface on all subclasses.

• You need to combine several behaviors, simulating multiple inheritance.
Exception Handling

One golden rule of programming:

**Errors Occur in Software Programs**

But what really matters is what happens after the error occurs. How is the error handled? Who handles it? Can the program recover?

An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions (out of bound array elements, division by zero).

Example:

InputFile.java:9:Exception java.io.FileNotFoundException must be caught, or declared in the throw clause of this method

```java
f = new FileInputStream(filename);
```

This indicates that the compiler found an exception that is not being dealt with.

Many kinds of errors can cause exceptions - problems ranging from serious hardware errors, such as a hard disk crash, to simple programming errors, such as trying to access an out-of-bounds array element.

When such an error occurs within a Java method, the method creates an exception object and hands it off to the runtime system.

The exception object contains information about the exception, including its type and the state of the program when the error occurred.

The runtime system is then responsible for finding some code to handle the error. In Java terminology, creating an exception object and handling it to the runtime system is called **throwing and exception**.
After a method throws an exception, the runtime system leaps into action to find someone to handle the exception. The set of possible “someones” to handle the exception is the set of methods in the call stack of the method where the error occurred.

The runtime system searches backwards through the call stack, beginning with the method in which the error occurred, until it finds a method that contains an appropriate exception handler. An exception handler is considered appropriate is the type of the exception thrown is the same as the type of exception handled by the handler.

The exception moves up through the call stack until an appropriate handler is found and one of the calling methods handles the exception. The exception handler chosen is said to catch the exception.

If the runtime system exhaustively searches all of the methods on the call stack without finding an appropriate exception handler, the runtime system (and consequently, the Java program) terminates.

Advantages over traditional error management techniques:

I. Separating Error Handling Code from “Regular” Code

```java
readFile() {
    open
    get size
    alloc mem
    read file
    close file
    What happens if ?
    can’t open
    can’t determine
    not enough mem
    read fails
    can’t close
}
```

To answer these questions within your function you’d have to add a lot of code to do error detection, reporting, and handling.
This technique has the following disadvantages:

- Significant increase in code. In the specific example it increases from 7 to 29 lines of code (400%).
- Original code lost in clutter.
- Logical flow of code lost in clutter.
- Hard to understand, read, etc.

Exceptions enable you to write the main flow of you code and deal with the exceptional cases elsewhere.

```java
readFile() {
    try {
        open
        get size
        allocate memory
        read file
        close file
    } catch (fileOpenedFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```
II. Propagating Errors Up the Call Stack

Suppose that the `readFile` method is the fourth in a series of nested method calls made by your main program: `m1` calls `m2`, which calls `m3`, which finally calls `readFile`.

```
method1 {
    call method2;
}
method2 {
    call method3;
}
method3 {
    call readFile;
}
```

Suppose that `method1` is the only method interested in the errors that might occur within `readFile`. Traditional error notification techniques force `method2` and `method3` to propagate the error codes returned by `readFile` up the call stack until the error codes finally reach `method1`.

The Java runtime system searches backwards through the call stack to find any methods that are interested in handling a particular exception.

A Java method can “duck” any exceptions thrown within it, thereby allowing a method further up the call stack to catch it. As a result only the methods that care about errors have to worry about detecting errors.
Ducking an exception requires specification. Any checked exceptions that can be thrown within a method are part of that method’s public programming interface and must be specified in the `throws` clause of the method.

A method informs its callers about the exceptions that it can throw, so that the callers can intelligently and consciously decide what to do about those exceptions.
III. Grouping and Differentiating Error Types

Often exceptions fall into categories or groups. You could imagine a group of exceptions concerning errors when manipulating arrays: index out of range, wrong type of element, etc…

All exceptions thrown in Java are objects, so grouping or categorization of exceptions is a natural outcome of the class hierarchy.

In some instances you might want some methods to all exceptions that fall within a category (all array exceptions)

```
    catch (ArrayException e) { . . .
```

and other methods to handle specific exceptions (invalid index exceptions)

```
    catch (InvalidIndexException e) { . . .
```

The Java language requires that methods either catch or specify all checked exceptions that can be thrown within the scope of that method. Otherwise an error message will be issued during compilation.

A method can catch an exception by providing an exception handler for that type of exception.

If a method chooses not to catch an exception, the method must specify that it can throw that exception. Why?
Because any exception that can be thrown by a method is really part of the method's public programming interface. Callers of a method must know about the exception that a method can throw in order to intelligently and consciously decide what to do about those exceptions. Thus, in the method signature, you specify the exceptions that the method can throw.

Java has different types of exceptions, including I/O exceptions, runtime exceptions, exceptions of your own creation, and so on. Runtime exceptions are those that occur within the Java runtime system. This includes arithmetic exceptions (dividing by zero), pointer exceptions (accessing an object with a null reference), and indexing exceptions (out of range index).

Runtime exceptions can occur anywhere in a program and in a typical program can be very numerous. Thus, the compiler does not require that you catch or specify runtime exceptions, although you can.

Example: ListOfNumbers

If you try to compile the ListOfNumbers class, the compiler prints an error message about the exception thrown by the FileOutputStream constructor, but does not display one about the exception thrown by elementAt.
The try Block

The first step in constructing an exception handler is to enclose the statements that might throw an exception within a try block. For the method writeList in the ListOfNumbers class there is more than one way to accomplish this task:

1. Put each statement that might potentially throw an exception within its own try statement and provide separate exception handlers.

2. Put all the statements within a single try block and associate multiple handlers with it (the code tends to be easier to read).

```java
public void writeList() {
    PrintStream pStr;

    try {
        int i;

        System.out.println("Entering Try");
        pStr = new PrintStream ( new BufferedOutputStream( new FileOutputStream("outFile.txt") ));

        for (i=0; i<size; i++)
            pStr.println( "value at: " + i + " = " + victor.elementAt(i) );

    }
}
```
The **catch** Block(s)

The **try** statement defines the scope of its associated exception handlers. You associate exception handlers with a **try** statement by providing one or more **catch** blocks directly after (no intervening code) the **try** block:

```
try {
    ...
} catch (aThrowableObj varName)
    Java statements
} catch (aThrowableObj varName)
    Java statements
}...
```

*aThrowableObj* is the type of exception that the handler can handle, *varName* is the name by which we can refer to the exception caught. You can access the instance variables and methods of exceptions in the same manner that you access those of other objects.

**Example:**

```
try {
    ...
} catch(ArrayIndexOutOfBoundsException e) {
    System.err.println("Array out of Bounds Exception: " + e.getMessage());
} catch(IOException e) {
    System.err.println("Caught an IO Exception: " + e.getMessage());
}
```
Catching Multiple Exceptions with One Handler

try { ... }

} catch (Exception e) {
    System.err.println("Caught an Exception: " + e.getMessage());
}

Generally speaking, your exception handler should be more specialized.

Handlers that can catch most or all exceptions are typically useless for error recovery because it has to determine what type of exception occurred anyway to determine the best recovery strategy.

Exception handlers that are too general can make code more error prone by catching exceptions that weren't anticipated by the programmer and for which the handler was not intended.

The finally block

The final step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program.

The runtime system always executes the statements within the finally block regardless of what happens within the try block.

        finally {
            if (pStr != null) {
                System.out.println("Closing PS");
                pStr.close();
            } else {
                System.out.println("PS not open");
            }
        }
Specifying the Exceptions Thrown by a Method

Sometimes it's appropriate for your code to catch exceptions that can occur within it. In other cases, however, it's better to let method further up the call stack to handle the exception.

If the method doesn't catch the exceptions that can occur within it, then it must specify that it can throw them.

Example:

```java
public writeList() throws IOException,
                ArrayIndexOutOfBoundsException
        {
            ...  
        }
```

How to Throw Exceptions

```java
public Object pop() throws EmptyStackException
{
    Object obj;

    if (size == 0) throw new EmptyStackException();

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size --;
    return obj;
}
```
Applets

An applet is a kind of program that is designed to be run within a Java-aware environment like a Web browser. An applet, unlike a Java application, isn’t a traditional stand-alone program, but rather is a class that is loaded and executed by another program.

Example:

```java
class Hello extends Applet {

    public void paint(Graphics g) {
        g.drawString("Hi", 20, 10);
    }
}
```

When an applet is loaded, here’s what happens:

1. An instance of the applet’s controlling class is created.
2. The applet initializes itself.
3. The applet starts running.
4. When the user leaves the page the applet has the option of stopping itself.
5. When the user returns to the page, the applet can start itself again. The same sequence occurs when the user closes and reopens the window containing the applet.
6. Some browsers let the user reload applets, which consists of unloading the applet and then loading it again. Before an applet is unloaded, it is given the chance to stop itself and then perform a **final cleanup**, so that the applet can release any resources it holds.

**Methods**

**init()**

To initialize the applet each time it is loaded. If not overloaded, it does nothing. Useful for one-time initialization that doesn’t take very long.

In general, it should contain the code that you would normally put into a constructor. The reason applets shouldn’t usually have constructors is that an applet isn’t guaranteed to have a full environment until its init method is called.

For example, the Applet image loading methods simply don’t work inside an applet constructor. The init method, on the other hand, is a great place to call the image loading methods.

**start()**

To start the applet’s execution, such as when the applet is loaded or when the user revisits a page that contain the applet. The Applet class does nothing in this method. The subclass should override this method to start animation, sound, and so on. Every applet that does something after initialization (except in direct response to user actions) must override this method. This method either performs the applet’s work or (more likely) starts up one or more threads to perform the work.

**stop()**

To stop the applet’s execution, for example, when the user leaves the applet’s page or quits the browser. The Applet class does nothing in this method. Most applets that override start should also override this method. The stop method should suspend the applet’s execution so that it doesn’t take up system resources when the user isn’t viewing the applet’s page.
**destroy()**

To perform final cleanup in preparation for unloading. The Applet class does nothing in this method. Multithreaded applets can use this method to stop any live threads. Most applets don’t need to override this method, since their stop (which is called before destroy) does everything necessary to shut down an applet’s execution. However, it is available for applets that need to release additional resources.

**Methods for Drawing**

**paint(Graphics g)**

The basic display method. It is called when the applet drawing area must be refreshed. The Applet class simply draws a gray box in the area. The subclass should override this method to actually draw something on the screen.

When an applet is initially executed, the paint method is automatically called (after **init** and **start**). For paint to be called again, an event, such as the user resizing the applet, must occur.

**update(Graphics g)**

This is the method that is called when the screen must be updated. The Applet class simply clears the area and calls **paint**. The update method is seldom called directly and sometimes overridden. Overriding the method is useful to improve drawing performance such as reducing flicker of animations.

**repaint()**

If the programmer needs to call paint, a call is made to the Component class method repaint. This method requests a call to the **update** method, and then update calls **paint** directly. The method repaint is frequently called by the programmer to force a paint operation. This method should not be overridden because it performs some system-dependent tasks.
What Applets Can and Can Not Do

Each browser implements security policies to keep applets from compromising system security.

Current browsers impose the following restrictions on any applet that is loaded over the network (An applet can not . . .):

- Load libraries or define native methods. Applets can only use their own Java code and the Java API the applet viewer provides.

- Read or write files on the host that is executing it. Applets in any applet viewer can read files specified with full URLs, instead of by a file name. A workaround for not being able to write files is to have the applet forward data to an application on the host the applet came from. This application can write the data files on its own host.

- Make network connections except to the host that it came from. The workaround is to have the applet work with an application on the host it came from. The application can make its own connections anywhere on the network.

- Start any program on the host that is executing it. Again, an applet can work with a server-side application instead.

- Read certain system properties

- Windows that an applet brings up look different from windows that an application brings up

The java.applet package provides an API that gives applets some capabilities that applications don't have. For example, applets can play sounds, which other programs aren't able to do.
Here are some other things that current browsers and other applet viewers let applets do:

- Applets can usually make network connections to the host they came from.

- Applets running within a Web browser can easily cause HTML documents to be displayed.

- Applets can invoke public methods of other applets on the same page.

- Applets that are loaded from the local file system (directory in the user's CLASSPATH) have none of the restrictions that applets loaded over the network do.

- Although most applets stop running once you leave their page, they don't have to.

**Graphical User Interface Components**

GUIs are built from GUI components (widgets) which are visual objects with which the user interacts via the mouse or the keyboard.

- **Label**: To display uneditable text
- **Button**: Triggers an event when clicked.
- **TextField**: To input data from the keyboard.
- **Choice**: A drop-down list of items
- **Checkbox**: Check boxes and radio buttons
- **List**: Menu of items
- **Panel**: A container for components
GUI Event Handling

An event is an action initiated by the user of the program interacting with the GUI such as the user moving the mouse, clicking the mouse, clicking a button, etc.

Events are sent to a Java program by the underlying windowing system. Information about a GUI event is stored in an object from a subclass of AWTEvent.

To process an event, the programmer must perform two key tasks – register an event listener and implement an event handler.

An event listener is an object of a class that implements an event-listener interface.

An event handler is a method that is automatically called in response to a particular type of event. Each event listener interface provides one or more event handling methods.

The use of event listeners in event handling is known as the delegation event model – the processing of an event is delegated to a particular object in the program.

Example: Push Buttons

A button is a component the user clicks to trigger a specific action, or generate an event.

Buttons generate “action” events. An action event can be processed by any ActionListener object (an object of any class that implements that interface).

The class must provide a definition of method actionPerformed. Once an ActionListener is “registered” to “handle” an action event, this method is called automatically whenever an action event occurs.

```java
b1 = new Button("aLabel");
b1.addActionListener( new Handler1(this));
```
Defines an object $b1$ of class **Button**, and then registers an object of the class **Handler1** as an **ActionListener** for the button.

Class **Handler1** must be a class that implements the interface **ActionListener**.

Method **addActionListener** can be found in every GUI component class that generates action events. If an event listener is not registered for a particular component, then no events are handled for that component.

Using a separate class to define an event listener is a common programming practice for separating the GUI interface from the implementation of its event handler.

**Example:**

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

public class MyButtons extends Applet {

    private Button pushButton1, pushButton2;

    public void init() {
        pushButton1 = new Button( "Button 1" );
        pushButton1.addActionListener(new Button1Handler(this) );
        add( pushButton1 );

        pushButton2 = new Button( "Button 2" );
        pushButton2.addActionListener(new Button2Handler(this) );
        add( pushButton2 );
    }
}
```
class Button1Handler implements ActionListener {

    Applet applet;

    public Button1Handler( Applet a )
    {
        applet = a;
    }

    public void actionPerformed( ActionEvent e )
    {
        applet.showStatus( "You pressed: " + e.getActionCommand() );
    }
}


class Button2Handler implements ActionListener {

    Applet applet;

    public Button2Handler( Applet a )
    {
        applet = a;
    }

    public void actionPerformed( ActionEvent e )
    {
        applet.showStatus( "You pressed: " + e.paramString() );
    }
}
Mouse Events

Mouse events can be trapped for any GUI component that derives from Component. The interfaces to handle mouse events are MouseListener and MouseMotionListener. The methods of these interfaces are:

- mousePressed(MouseEvent e)
- mouseClicked(MouseEvent e)
- mouseReleased(MouseEvent e)
- mouseEntered(MouseEvent e)
- mouseExited(MouseEvent e)
- mouseDragged(MouseEvent e)
- mouseMoved(MouseEvent e)

The MouseEvent object contains information about the mouse event that occurred including the x and y coordinates where the event occurred.

Example: Mouse Tracker

The applet uses xPos and yPos to store the location of the mouse when a mouse event occurs.

The strings output from the paint method are always displayed at these coordinates.

The applet’s init method registers the applet as the listener for all mouse events:

- addMouseListener(this)
- addMouseMotionListener(this)

These are Component methods that can be used to register mouse event listeners for any object of any class that extends Component.
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class MouseTracker extends Applet
    implements MouseListener, MouseMotionListener {

    private int xPos, yPos = -10; // off screen for 1st paint
    private String s = "";

    public void init()
    {
        // applet listens to its own mouse events
        addMouseListener( this );
        addMouseMotionListener( this );
    }

    public void paint( Graphics g )
    {
        g.drawString( s + " @ [" + xPos + "," + yPos + "]", xPos, yPos );
    }

    private void setValues(String event, int x, int y)
    {
        s = event;
        xPos = x;
        yPos = y;
        repaint();
    }
}
// MouseListener event handlers

public void mouseClicked( MouseEvent e ) {
    setValue("Clicked", e.getX(), e.getY());
}

global void mousePressed( MouseEvent e ) {
    setValue("Pressed", e.getX(), e.getY());
}

global void mouseReleased( MouseEvent e ) {
    setValue("Released", e.getX(), e.getY());
}

global void mouseEntered( MouseEvent e ) {
    showStatus("Mouse in applet area");
}

global void mouseExited( MouseEvent e ) {
    showStatus("Mouse outside applet area");
}

// MouseMotionListener event handlers

public void mouseDragged( MouseEvent e ) {
    setValue("Dragging", e.getX(), e.getY());
}

global void mouseMoved( MouseEvent e ) {
    setValue("Moving", e.getX(), e.getY());
}
**Adapter Classes**

Many of the event-listener interfaces provide multiple methods. However, it is not always desirable to define every method in an event – listener interface.

For example, a program may only need the `mouseClicked` handler of the `MouseListener` interface.

For this reason, the `java.awt.event` package provides the event-listener adapter classes. Every event-listener interface with more than one method has a corresponding adapter class that provide a default implementation of every method in the interface.

The programmer can extend the adapter class to inherit the default implementation and simply override the method or methods needed for event handling in the program. The default implementation has an empty body.

**Example:** Draw pictures with the mouse. Defines class `MotionHandler` as a subclass of `MouseMotionAdapter` and overrides the `mouseDragged` method.

The instance variables `xValue` and `yValue` store the coordinates of the `mouseDragged` event.

An object of `MotionHandler` is registered to listen for the applet’s mouse motion events:

```
addMouseMotionHandler(new MotionHandler(this))
```

The constructor receives a reference to the applet to enable the event handler to interact with the applet when an event occurs.

The `update` method is overridden to prevent clearing the background when `repaint` is called. Normally, `update` draws a filled rectangle the size of the applet in the current background color, then calls `paint`. The new version simply calls `paint` directly.
import java.applet.Applet, java.awt.*, java.awt.event.*;

public class Drag extends Applet {
    private int xValue = -10, yValue = -10;

    public void init() {
        addMouseMotionListener(new MotionHandler(this));
    }

    public void paint( Graphics g ) {
        g.drawString( "Drag the mouse to draw", 10, 20 );
        g.fillOval( xValue, yValue, 4, 4 );
    }

    public void update( Graphics g )
    { paint( g ); }

    public void setCoordinates( int x, int y ) {
        xValue = x;
        yValue = y;
        repaint();
    }

    class MotionHandler extends MouseMotionAdapter {
        private Drag dragger;

        public MotionHandler(Drag d)
        { dragger = d; }

        public void mouseDragged(MouseEvent e) {
            dragger.setCoordinates(e.getX(), e.getY());
        }
    }
}
The **MouseDetails** applet demonstrates how to determine the number of mouse clicks and how to distinguish between different mouse buttons.

Class **MouseEvent** inherits several methods from class **InputEvent** to distinguish between mouse buttons: **isMetaDown()**, **isAltDown()**.

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

public class MouseDetails extends Applet {

    private String s = "";
    private int xPos, yPos;

    public void init() {
        addMouseListener(new DetailHandler(this));
    }

    public void paint(Graphics g) {
        g.drawString("Clicked @ [" + xPos + ", " + yPos + "]", xPos, yPos);
    }

    public void setCoordinates( int x, int y ){
        xPos = x;
        yPos = y;
        repaint();
    }
}
```
class DetailHandler extends MouseAdapter {

    private MouseDetails details;

    public DetailHandler( MouseDetails m )
    {
        details = m;
    }

    public void mouseClicked( MouseEvent e )
    {
        details.setCoordinates( e.getX(), e.getY() );

        String s = "Clicked " + e.getClickCount() + " time(s)";

        if ( e.isMetaDown() ) // Right button
            s += " with right mouse button";
        else if ( e.isAltDown() ) // Middle button
            s += " with center mouse button";
        else // Left button
            s += " with left mouse button";

        details.showStatus( s );
    }
}
// Demonstrating keystroke events.

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

public class Key extends Applet implements KeyListener {
    private String line1 = "";
    private String line2 = "";
    private String line3 = "";

    public void init()
    {
        // allow applet to process Key events
        addKeyListener( this );

        // make applet the active component for key events
        requestFocus();
    }

    public void paint( Graphics g )
    {
        g.drawString( line1, 25, 25 );
        g.drawString( line2, 25, 40 );
        g.drawString( line3, 25, 55 );
    }

    public void keyTyped( KeyEvent e )
    {
        line1 = "Key typed: " + e.getKeyChar();
        setLines2and3( e );
    }
}

public void setLines2and3( KeyEvent e )
{
    line2 = "" + e.getKeyChar();
    line3 = "" + e.getKeyChar();
}
public void keyPressed( KeyEvent e ) {
    line1 = "Key pressed: " +
            e.getKeyText( e.getKeyCode() );
    setLines2and3( e );
}

public void keyReleased( KeyEvent e ) {
    line1 = "Key released: " + e.getKeyText( e.getKeyCode() );
    setLines2and3( e );
}

private void setLines2and3( KeyEvent e ) {

    line2 = "This key is " + ( e.isActionKey() ? "" : "not " ) +
            "an action key";

    String temp =  e.getKeyModifiersText( e.getModifiers() );

    line3 = "Modifier keys pressed: " +
            ( temp.equals( "" ) ? "none" : temp );

    repaint();
}