Interface

1. Interface

An interface is a reference type in Java. It is similar to "class". It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways.

(1) An interface can contain any number of methods.
(2) An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.

However, an interface is different from a class in several ways.

(a) We cannot instantiate an interface.
(b) An interface does not contain any constructors.
(c) All of the methods in an interface are abstract.
(d) An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
(e) An interface is not extended by a class; it is implemented by a class.
(f) An interface can extend multiple interfaces.

The syntax of interface is as follows.

```
public interface <interface_name>{
    //methods
}
```

An interface is implicitly abstract. You do not need to use the **abstract** keyword while declaring an interface. Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.

To use an interface in the class, append the keyword "implements" after the class name followed by the interface name. For example

```
public class <class_name> implements <interface_name> {
}
```

To understand the concept of Java Interface better, let see an example. Suppose we have a requirement where class "Dog" inheriting class "Animal" and "Pet". But we cannot extend two classes in Java. However, class can implement any number of interfaces. The one of solution is that "Dog" class extends to "Animal" class and implements interface as "Pet" class.

```
public class Dog extends Animal implements Pet {
}
```

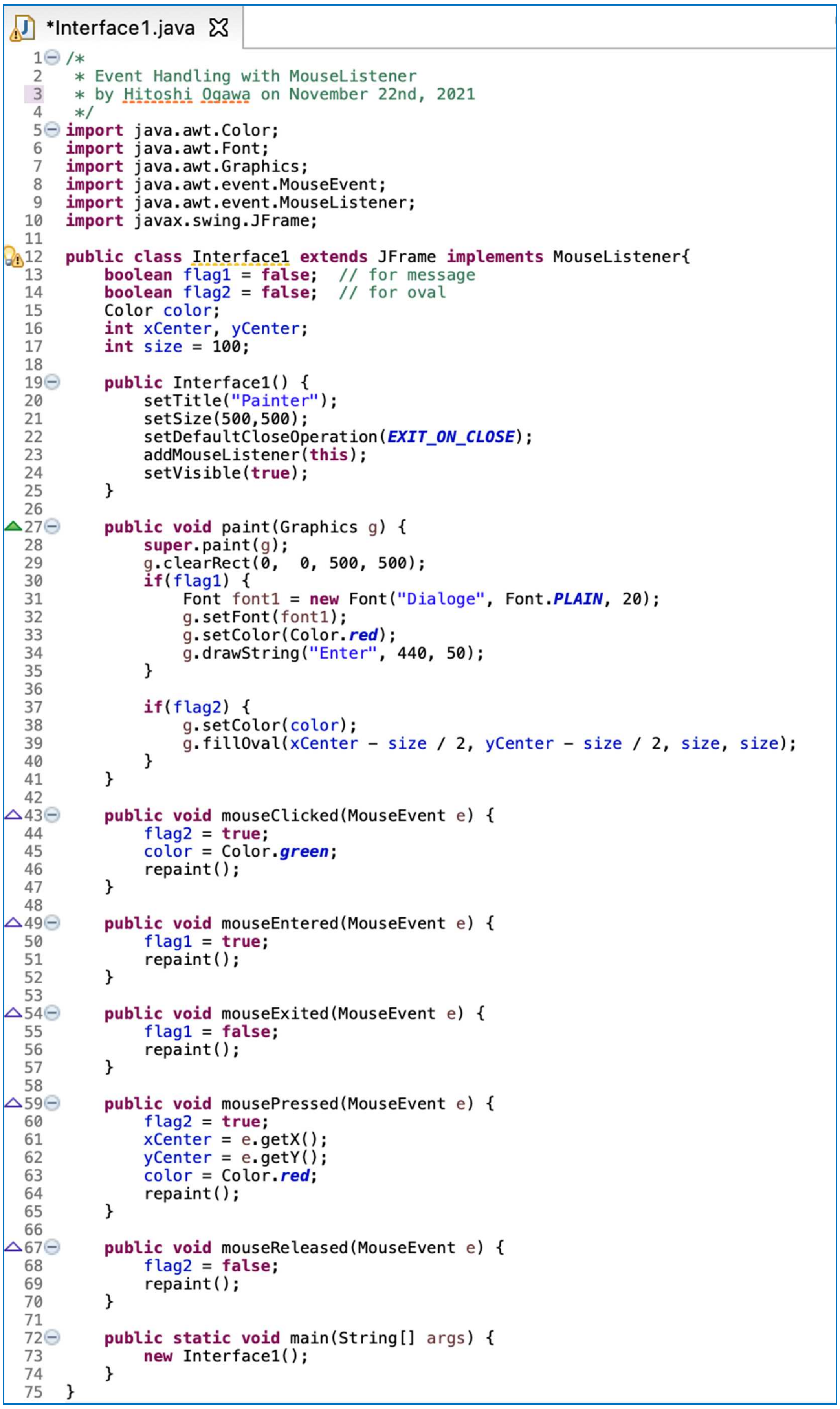
2. MouseListener

To implement event processing, "MouseAdapter" and "MouseMotionAdapter" were usen in "AWT Event Handling" (The sixth day). Since these are classes, they had to be implemented as inner classes. However, if we use a listener, we can implement event processing in a class without using an inner class.

MouseListener is the listener interface for receiving mouse event (press, release, click, enter and exit) on a component. MouseListener provides five abstract methods, so we need to implement them. Figure 1 shows the program "Interface1" that inherits JFrame and implements MouseListener as an example.

When the mouse cursor enters the window, the string "Enter" is posted on the upper right (Figure 2 (a)). When the mouse cursor moves outside the window, the string disappears (Figure 2 (b)). To post and erase string, the Boolean variable "flag1" is used. In mouseEntered method, flag1 is substituted for true and repaint method is executed (lines 50 and 51). When flag1 is true, the string is displayed, otherwise it is not displayed (lines 30 to 35). In mouseExsited method, flag1 is substituted for false and repaint method is executed (lines 55 and 56).

A red circle with a size of 100 pixels is filled when the mouse is pressed in the window (Figure 2 (c)), and it disappears when released. However, (the mouse is released in the same place as the pressed place) (Figure 2 (d)). Whether a circle is displayed or not can be specified by the variable "flag2". If the value of flag2 is true, the circle is filled with the color specified by the variable "color". In the mousePressed method, flag2 is assigned true, and the coordinates of the pressed place are assigned to xCenter and yCenter. "Color.red" is assigned to the variable color and repaint method is executed (lines 60 to 64). In the mouseReleased method, flag2 is assigned false, and repaint method is executed (lines 68 and 69). In the mouseClicked method, flag2 is assigned true, "Color.green" is assigned to the variable color, and repaint method is executed (lines 44 to 46). The mouseClicked runs effectively, because when the click event occurs it will be after the release event.



*Interface1.java

```java
/*
 * Event Handling with MouseListener
 * by Hitoshi Ogawa on November 22nd, 2021
 */
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JFrame;

public class Interface1 extends JFrame implements MouseListener{
    boolean flag1 = false;  // for message
    boolean flag2 = false;  // for oval
    Color color;
    int xCenter, yCenter;
    int size = 100;

    public Interface1() {
        setTitle("Painter");
        setSize(500,500);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        addMouseListener(this);
        setVisible(true);
    }

    public void paint(Graphics g) {
        super.paint(g);
        g.clearRect(0,  0, 500, 500);
        if(flag1) {
            Font font1 = new Font("Dialoge", Font.PLAIN, 20);
            g.setFont(font1);
            g.setColor(Color.red);
            g.drawString("Enter", 440, 50);
        }

        if(flag2) {
            g.setColor(color);
            g.fillOval(xCenter - size / 2, yCenter - size / 2, size, size);
        }
    }

    public void mouseClicked(MouseEvent e) {
        flag2 = true;
        color = Color.green;
        repaint();
    }

    public void mouseEntered(MouseEvent e) {
        flag1 = true;
        repaint();
    }

    public void mouseExited(MouseEvent e) {
        flag1 = false;
        repaint();
    }

    public void mousePressed(MouseEvent e) {
        flag2 = true;
        xCenter = e.getX();
        yCenter = e.getY();
        color = Color.red;
        repaint();
    }

    public void mouseReleased(MouseEvent e) {
        flag2 = false;
        repaint();
    }

    public static void main(String[] args) {
        new Interface1();
    }
}
```

Figure 1. An example program that inherits JFrame and implements MouseListener.

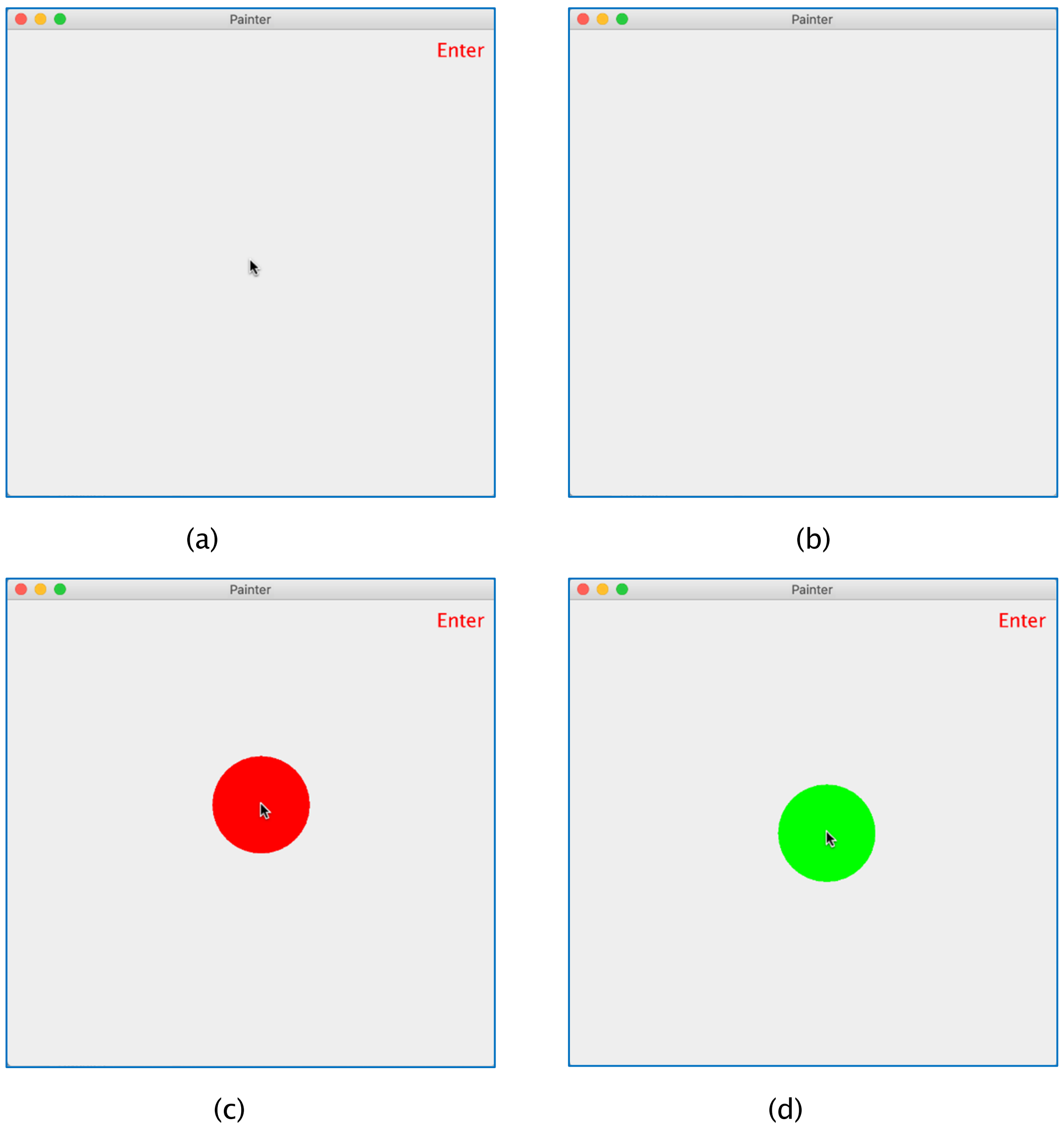


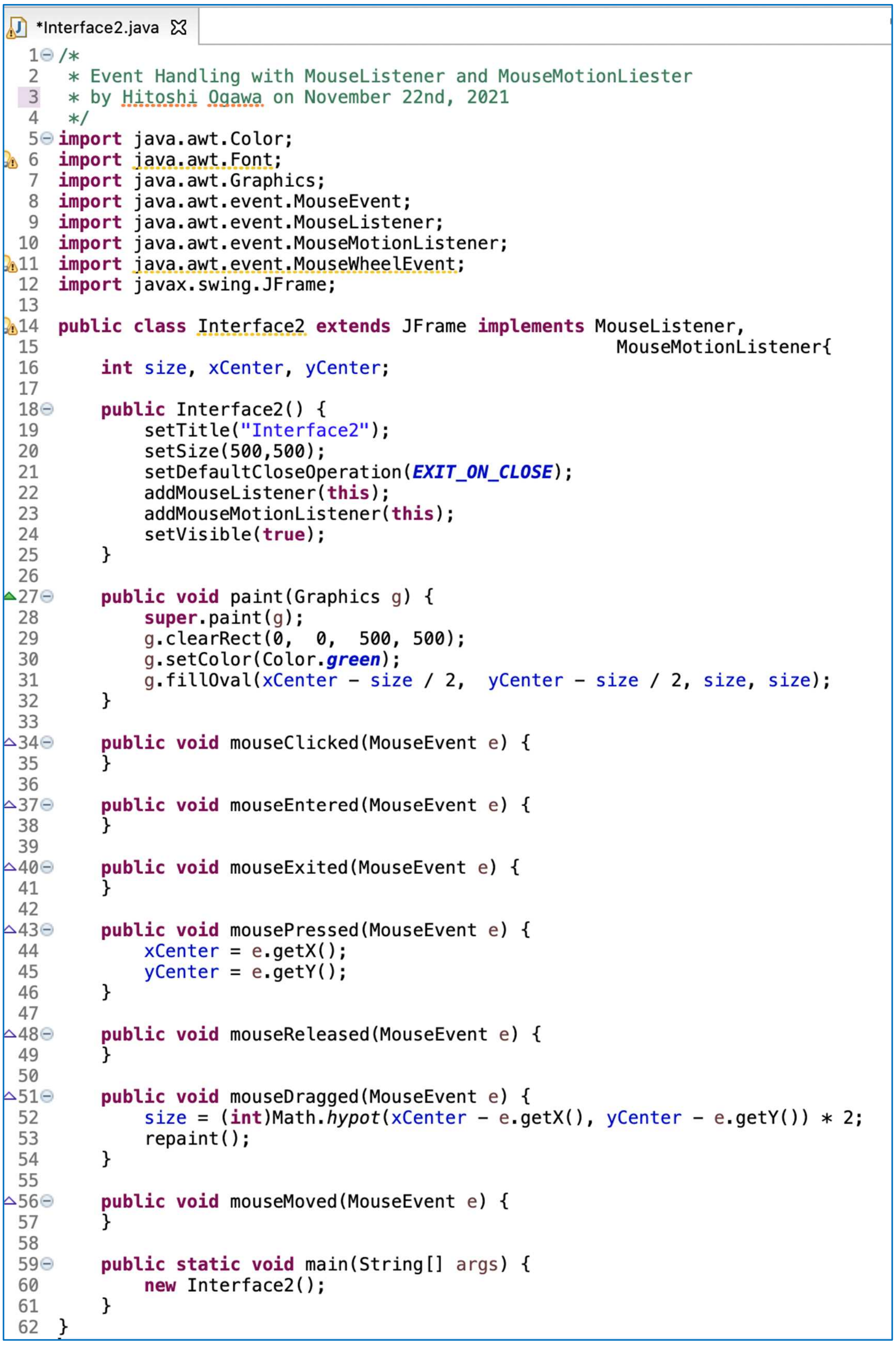
Figure 2. (a) The mouse cursor enters the window,
(b) The mouse cursor moves outside the window,
(c) The mouse is pressed in the window,
(d) The mouse is clicked.

3. MouseMotion and MouseMotionListener

Figure 3 shows the program “Interface2” realizing “MouseMotion” introduced on “AWT Event Handling” (The sixth day) using a listener. Since mousePressed method and mouseDragged method are used, MouseListener and MouseMotionListener must be implemented.

MouseMotionListener provides two abstruct methods (mouseDragged(MouseEvent e), mouseMoved(MouseEvent e)).

We only need to use one from each listener, but we need to implements all the methods in the program “Interface2”.



```
/*
 * Event Handling with MouseListener and MouseMotionLiester
 * by Hitoshi Ogawa on November 22nd, 2021
 */
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseWheelEvent;
import javax.swing.JFrame;

public class Interface2 extends JFrame implements MouseListener,
                                                    MouseMotionListener{
    int size, xCenter, yCenter;

    public Interface2() {
        setTitle("Interface2");
        setSize(500,500);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        addMouseListener(this);
        addMouseMotionListener(this);
        setVisible(true);
    }

    public void paint(Graphics g) {
        super.paint(g);
        g.clearRect(0,  0,  500, 500);
        g.setColor(Color.green);
        g.fillOval(xCenter - size / 2,  yCenter - size / 2, size, size);
    }

    public void mouseClicked(MouseEvent e) {
    }

    public void mouseEntered(MouseEvent e) {
    }

    public void mouseExited(MouseEvent e) {
    }

    public void mousePressed(MouseEvent e) {
        xCenter = e.getX();
        yCenter = e.getY();
    }

    public void mouseReleased(MouseEvent e) {
    }

    public void mouseDragged(MouseEvent e) {
        size = (int)Math.hypot(xCenter - e.getX(), yCenter - e.getY()) * 2;
        repaint();
    }

    public void mouseMoved(MouseEvent e) {
    }

    public static void main(String[] args) {
        new Interface2();
    }
}
```

Figure 3. Interface2.

The result of Interface2 is the same as the example shown in Figure 4 on page 10 of “AWT Event Handling” (The sixth day), which is also shown in Figure 4.

“MouseAdapter” and “MouseMotionAdapter” introduced in “AWT Event Handling” (The sixth day) are classes. So, we can override only the necessary methods in the class that inherits it. However, these classes often need to be implemented as inner class.

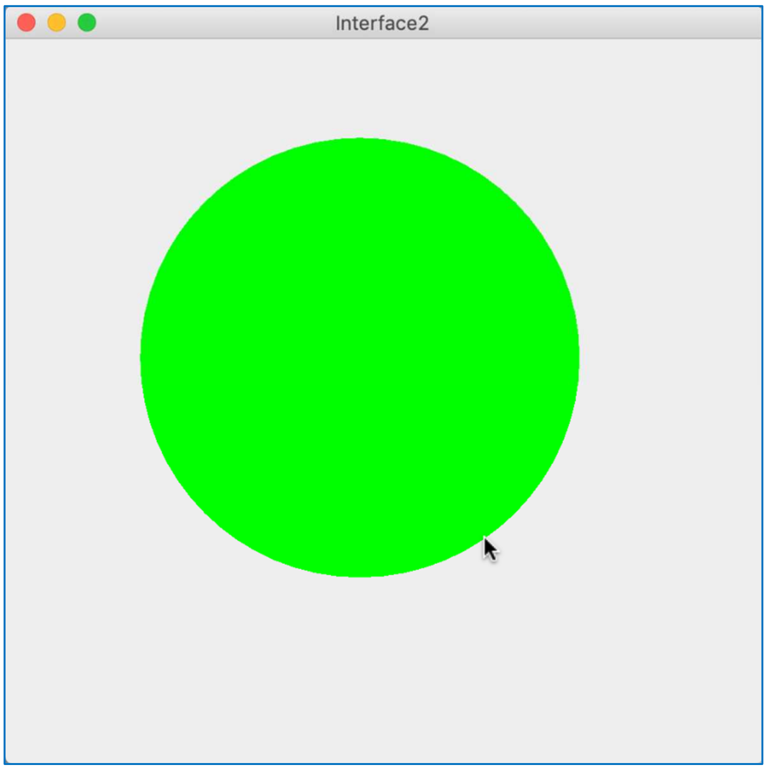


Figure 4. An example of executing Interface2

4. ActionListener

The interface “ActionListener” is notified whenever we click the button or menu item. It is notified against ActionEvent. The ActionListener has only one method: actionPerformed().

The object that implements the ActionListener interface gets the ActionEvent class when the event occurs. The main field variables that provide constants and methods are introduced below.

Field & Description of ActionEvent
int ALT_Mask The alt modifier. An indicator that the alt key was held down during the event.
int CTRL_MASK The control modifier. An indicator that the control key was held down during the event.
int SHIFT_MASK The shift modifier. An indicator that the shift key was held down during the event.

Method & Description of ActionEvent, and EventObject (which is the superclass of ActionEvent)
String getActionCommand() Returns the string identifying the command for this event.
int getModifiers() Returns the modifier keys held down during this action event.
Object getSource() Returns the object on which the Event initially occurred

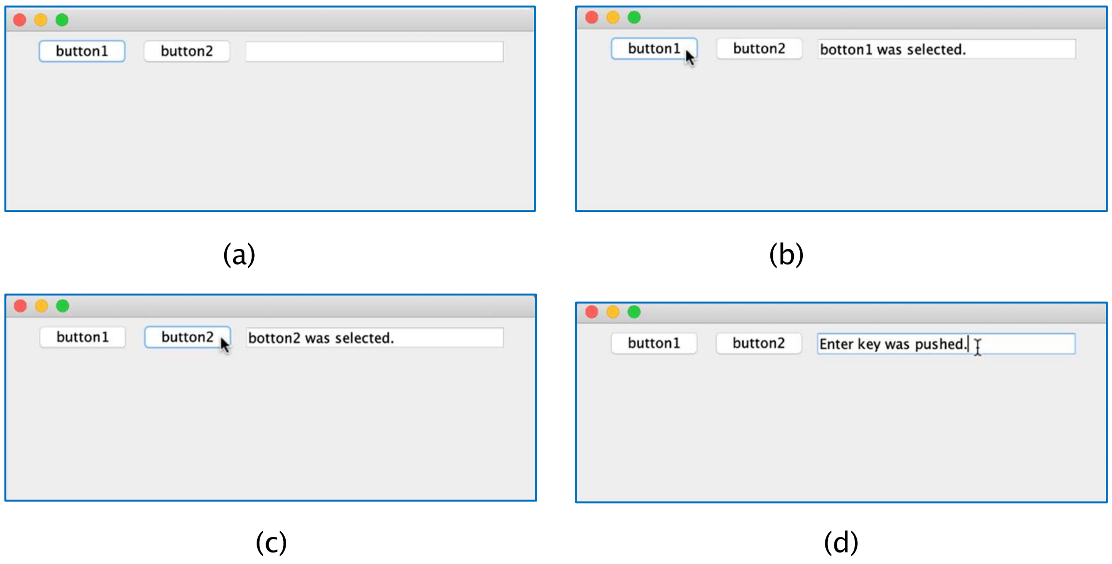
Two programs “Listener1” and “Listener2” that perform the operation shown in Figure 5 are introduced. Figure 5 (a) show the initial state. If press and Release at “button1”, the message “button1 was selected.” is displayed in the text field (Figure 5 (b)). If press and Release at “button2.”, the message “button2 was selected” is displayed in the text field (Figure 5 (c)). If “Enter key” is pushed after clicking the text field, the message “Enter key was pushed.” is displayed in the text field (Figure 5 (d)).

The program source of Listener1 is shown in Figure 6. Listener1 implements “ActionListener” (line 13). The variables “button1” and “button2” are assigned the JButton object named “button1” (line 22) and “button2” (line 25) respectively. The variable “text” is assigned the JTextField with 20 columns (line 28). And the ActionListener is added to all variables using method “addActionListener” (lines 23, 26 and 29). The keyword “this” is a reference variable that refers to the current object. In this case, “this” refers Listener1 class.

Since the “actionPerformed” method is activated regardless of which item the event occurs, it is necessary to know which item caused it. In the case of Listener1, the action command is used.

The command string is set using “setActionCommand” method of JButton and JTextField (lines 24, 27 and 30). The method “setActionCommand” was introduced at 4.2 (B) and 4.3 (B) of “Swing” (The seventh day). The command string is acquired using “getActionCommand” method of ActionEvent. In the case of JButton, an event occurs when pressing and releasing with same button. In the case of JTextField, an event occurs when pressing the Enter key.

The program source of Listener2 is shown in Figure 7. The getSource method of ActionEvent is used to acquire the object on which the Event initially occurred. Since the command strings are not handled, there is no need to use setActionCommand method. However, since it deals with the command string, it is suitable for dealing with large items. An example is introduced in Chapter 5.

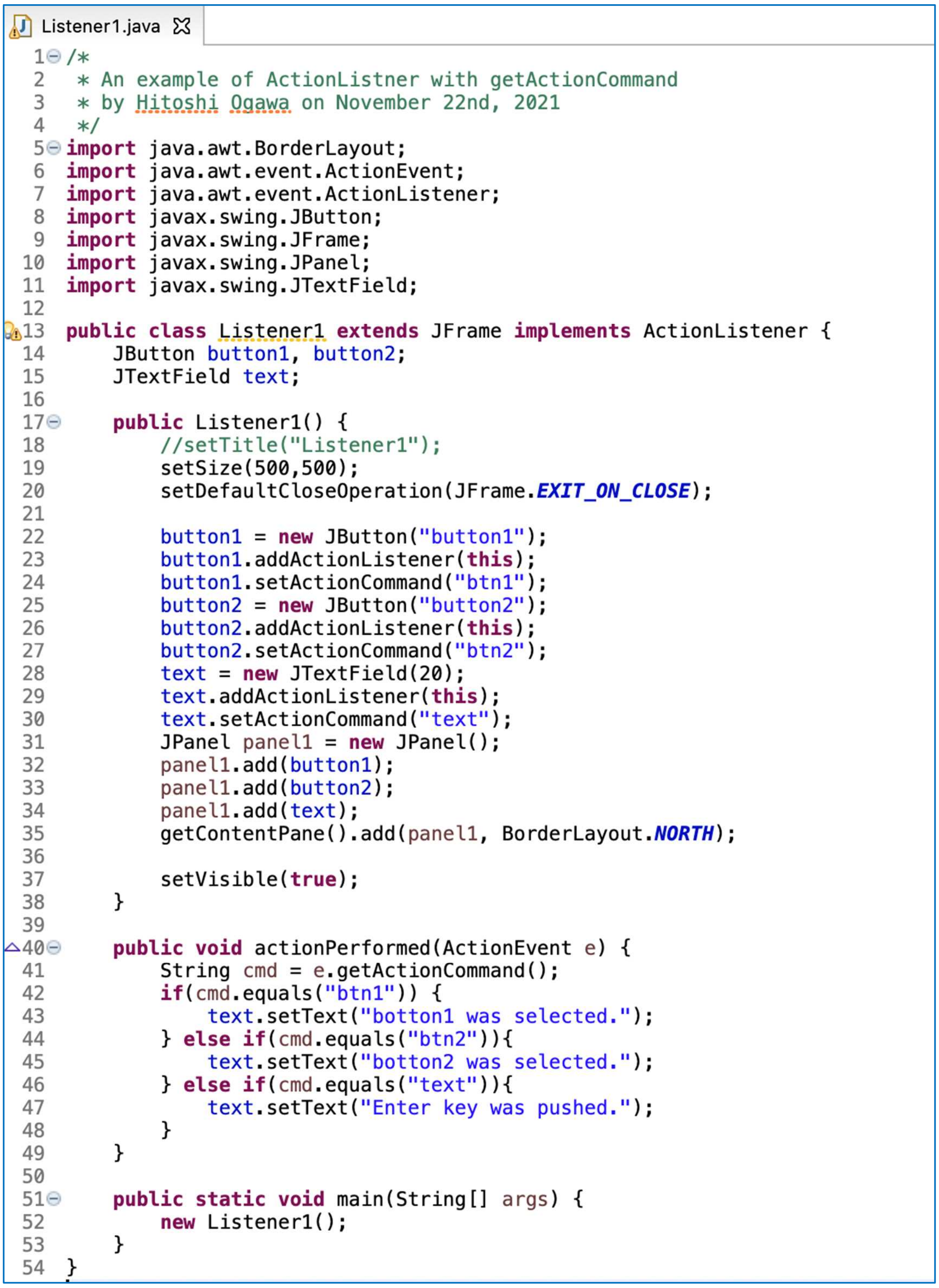


botton1 was selected.

botton2 was selected.

Enter key was pushed.

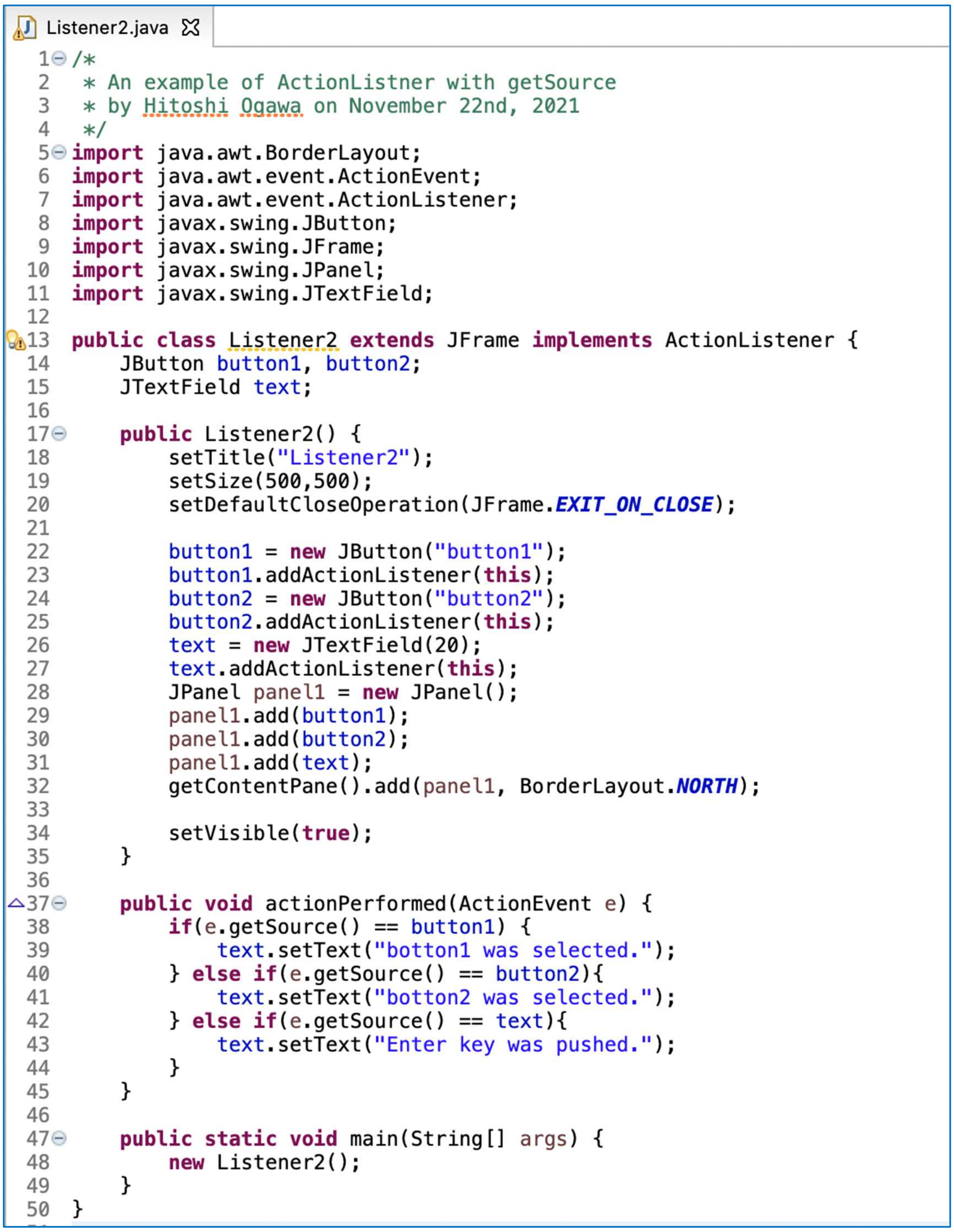
Figure 5. (a) The initial state, (b) When the “button1” is clicked,
(c) When the “button2” is clicked,
(d) Pressing the Enter key in the text area.



Listener1.java

```java
/*
 * An example of ActionListner with getActionCommand
 * by Hitoshi Ogawa on November 22nd, 2021
 */
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class Listener1 extends JFrame implements ActionListener {
    JButton button1, button2;
    JTextField text;

    public Listener1() {
        //setTitle("Listener1");
        setSize(500,500);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        button1 = new JButton("button1");
        button1.addActionListener(this);
        button1.setActionCommand("btn1");
        button2 = new JButton("button2");
        button2.addActionListener(this);
        button2.setActionCommand("btn2");
        text = new JTextField(20);
        text.addActionListener(this);
        text.setActionCommand("text");
        JPanel panel1 = new JPanel();
        panel1.add(button1);
        panel1.add(button2);
        panel1.add(text);
        getContentPane().add(panel1, BorderLayout.NORTH);

        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        if(cmd.equals("btn1")) {
            text.setText("botton1 was selected.");
        } else if(cmd.equals("btn2")){
            text.setText("botton2 was selected.");
        } else if(cmd.equals("text")){
            text.setText("Enter key was pushed.");
        }
    }

    public static void main(String[] args) {
        new Listener1();
    }
}
```

Figure 6. The program source of Listener1.



Listener2.java

```java
/*
 * An example of ActionListner with getSource
 * by Hitoshi Ogawa on November 22nd, 2021
 */
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class Listener2 extends JFrame implements ActionListener {
    JButton button1, button2;
    JTextField text;

    public Listener2() {
        setTitle("Listener2");
        setSize(500,500);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        button1 = new JButton("button1");
        button1.addActionListener(this);
        button2 = new JButton("button2");
        button2.addActionListener(this);
        text = new JTextField(20);
        text.addActionListener(this);
        JPanel panel1 = new JPanel();
        panel1.add(button1);
        panel1.add(button2);
        panel1.add(text);
        getContentPane().add(panel1, BorderLayout.NORTH);

        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        if(e.getSource() == button1) {
            text.setText("botton1 was selected.");
        } else if(e.getSource() == button2){
            text.setText("botton2 was selected.");
        } else if(e.getSource() == text){
            text.setText("Enter key was pushed.");
        }
    }

    public static void main(String[] args) {
        new Listener2();
    }
}
```

Figure 7. The program source of Listener2.

5. Switch Statement

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

The syntax is as follows.

```
switch(expression) {
    case valueA:
        // Processing when the value of the expression matches valueA.
        break;   // optional
    case valueB:
        // Processing when the value of the expression matches valueB.
        break;   // optional
    ...
    default :
        // Processing when none of the cases is true
}
```

The following rules apply to a switch statement:

- ✓ The variable used in a switch statement can only be integers and strings.
- ✓ We can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- ✓ The value for a case must be the same data type as the variable in the switch and it must be a constant or a literal.
- ✓ When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- ✓ When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- ✓ Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- ✓ A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

The actionPerformed method in Listner1 is rewritten using the switch statement as follows.

```
    public void actionPerformed(ActionEvent e) {
        String cmd = e.getActionCommand();
        switch(cmd) {
        case "btn1":
            text.setText("botton1 was selected.");
            break;
        case "btn2":
            text.setText("botton2 was selected.");
            break;
        case "text" :
            text.setText("Enter key was pushed.");
            break;
        }
    }
```