Chapter 11
Exceptions

Java Software Solutions
Foundations of Program Design
Seventh Edition

John Lewis
William Loftus

Copyright © 2012 Pearson Education, Inc.
Exceptions

• Exception handling is an important aspect of object-oriented design

• Chapter 11 focuses on:
  – the purpose of exceptions
  – exception messages
  – the try-catch statement
  – propagating exceptions
  – the exception class hierarchy
  – GUI mnemonics and tool tips
  – more GUI components and containers
Outline

Exception Handling
The try-catch Statement
Exception Classes
I/O Exceptions
Tool Tips and Mnemonics
Combo Boxes
Scroll Panes and Split Panes
Exceptions

• An exception is an object that describes an unusual or erroneous situation.

• Exceptions are thrown by a program, and may be caught and handled by another part of the program.

• A program can be separated into a normal execution flow and an exception execution flow.

• An error is also represented as an object in Java, but usually represents a unrecoverable situation and should not be caught.
Exception Handling

• The Java API has a predefined set of exceptions that can occur during execution

• A program can deal with an exception in one of three ways:

  1. ignore it
  2. handle it where it occurs
  3. handle it an another place in the program

• The manner in which an exception is processed is an important design consideration
Exception Handling

• If an exception is **ignored (not caught)** by the program, the program will terminate and produce an appropriate message

• The message includes a *call stack trace* that:
  – indicates the line on which the exception occurred
  – shows the method call trail that lead to the attempted execution of the offending line

• See Zero.java
public class Zero
{
    // Deliberately divides by zero to produce an exception.
    public static void main (String[] args)
    {
        int numerator = 10;
        int denominator = 0;

        System.out.println (numerator / denominator);
        System.out.println ("This text will not be printed.");
    }
}
public class Zero
{
    // Deliberately divides by zero to produce an exception.
    public static void main (String[] args)
    {
        int numerator = 10;
        int denominator = 0;

        System.out.println (numerator / denominator);

        System.out.println ("This text will not be printed.");
    }
}
Outline

Exception Handling
The try-catch Statement
Exception Classes
I/O Exceptions
Tool Tips and Mnemonics
Combo Boxes
Scroll Panes and Split Panes
The try Statement

• To handle an exception in a program, use a try-catch statement

• A try block is followed by one or more catch clauses

• Each catch clause has an associated exception type and is called an exception handler

• When an exception occurs within the try block, processing immediately jumps to the first catch clause that matches the exception type

• See ProductCodes.java
import java.util.Scanner;

public class ProductCodes
{
    public static void main (String[] args)
    {
        String code;
        char zone;
        int district, valid = 0, banned = 0;

        Scanner scan = new Scanner (System.in);

        System.out.print ("Enter product code (XXX to quit): ");
        code = scan.nextLine();

        continue
continue

    while (!code.equals("XXX"))
    {
        try
        {
            zone = code.charAt(9);
            district = Integer.parseInt(code.substring(3, 7));
            valid++;
            if (zone == 'R' && district > 2000)
                banned++;
        }
        catch (StringIndexOutOfBoundsException exception)
        {
            System.out.println("Improper code length: " + code);
        }
        catch (NumberFormatException exception)
        {
            System.out.println("District is not numeric: " + code);
        }

        System.out.print("Enter product code (XXX to quit): ");
        code = scan.nextLine();
    }

    System.out.println("# of valid codes entered: " + valid);
    System.out.println("# of banned codes entered: " + banned);
}
Sample Run

Enter product code (XXX to quit): TRV2475A5R-14
Enter product code (XXX to quit): TRD1704A7R-12
Enter product code (XXX to quit): TRL2k74A5R-11
District is not numeric: TRL2k74A5R-11
Enter product code (XXX to quit): TRQ2949A6M-04
Enter product code (XXX to quit): TRV2105A2
Improper code length: TRV2105A2
Enter product code (XXX to quit): TRQ2778A7R-19
Enter product code (XXX to quit): XXX
# of valid codes entered: 4
# of banned codes entered: 2

catch (NumberFormatException exception)
{
    System.out.println("District is not numeric: " + code);
}

System.out.print("Enter product code (XXX to quit): ");
code = scan.nextLine();
}

System.out.println("# of valid codes entered: " + valid);
System.out.println("# of banned codes entered: " + banned);
}
The finally Clause

- A try statement can have an optional **finally** clause, which **is always executed**

- **If no exception** is generated, the statements in the finally clause are executed after the statements in the try block finish

- **If an exception is generated**, the statements in the finally clause are executed after the statements in the appropriate catch clause finish
Exception Propagation

- An exception can be handled at a higher level if it is not appropriate to handle it where it occurs.

- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the level of the main method.

- See `Propagation.java`.

- See `ExceptionScope.java`.
public class Propagation
{
    //------------------------------------------------------------------------------
    // Invokes the level1 method to begin the exception demonstration.
    //------------------------------------------------------------------------------
    static public void main (String[] args)
    {
        ExceptionScope demo = new ExceptionScope();

        System.out.println("Program beginning.");
        demo.level1();
        System.out.println("Program ending.");
    }
}
public class Propagation {

// Invokes the level1 method to begin the exception demonstration.

static public void main (String[] args) {
    ExceptionScope demo = new ExceptionScope();
    System.out.println("Program beginning.");
    demo.level1();
    System.out.println("Program ending.");
}

Output
Program beginning.
Level 1 beginning.
Level 2 beginning.
Level 3 beginning.

The exception message is: / by zero

The call stack trace:
java.lang.ArithmeticException: / by zero
    at ExceptionScope.level3(ExceptionScope.java:54)
    at ExceptionScope.level2(ExceptionScope.java:41)
    at ExceptionScope.level1(ExceptionScope.java:18)
    at Propagation.main(Propagation.java:17)

Level 1 ending.
Program ending.
public class ExceptionScope
{
    public void level1()
    {
        System.out.println("Level 1 beginning.");

        try
        {
            level2();
        } catch (ArithmeticException problem)
        {
            System.out.println ();
            System.out.println ("The exception message is: " + problem.getMessage());
            System.out.println ();
            continue
        }
    }
}
System.out.println ("The call stack trace:");
problem.printStackTrace();
System.out.println ();

System.out.println("Level 1 ending.");

//  Serves as an intermediate level. The exception propagates
//  through this method back to level1.
//  -----------------------------------------------------------------
public void level2()
{
    System.out.println("Level 2 beginning.");
    level3 ();
    System.out.println("Level 2 ending.");
}

continue
public void level3 () {
    int numerator = 10, denominator = 0;

    System.out.println("Level 3 beginning.");
    int result = numerator / denominator;
    System.out.println("Level 3 ending.");
}

The Exception Class Hierarchy

• Exception classes in the Java API are related by inheritance, forming an exception class hierarchy

• All Error and Exception classes are descendents of the Throwable class

• A programmer can define an exception by extending the Exception class or one of its descendants

• The parent class used depends on how the new exception will be used
The Exception Class Hierarchy

- Object
  - Throwable
    - Error
      - LinkageError
      - ThreadDeath
      - VirtualMachineError
      - AWTError
    - Exception
      - RuntimeException
        - ArithmeticException
        - IndexOutOfBoundsException
        - NullPointerException
      - IllegalAccessException
      - NoSuchMethodException
      - ClassCastException
Checked Exceptions

• An exception is either checked or unchecked

• A checked exception must either be caught or must be listed in the throws clause of any method that may throw or propagate it

• A throws clause is appended to the method header

• The compiler will issue an error if a checked exception is not caught or listed in a throws clause
Unchecked Exceptions

• An unchecked exception does not require explicit handling, though it could be processed that way

• The only unchecked exceptions in Java are objects of type `RuntimeException` or any of its descendants

• Errors are similar to `RuntimeException` and its descendants in that:
  – Errors should not be caught
  – Errors do not require a throws clause
Quick Check

Which of these exceptions are checked and which are unchecked?

- NullPointerException
- IndexOutOfBoundsException
- ClassNotFoundException
- NoSuchMethodException
- ArithmeticException
Quick Check

Which of these exceptions are checked and which are unchecked?

- NullPointerException: Unchecked
- IndexOutOfBoundsException: Unchecked
- ClassNotFoundException: Checked
- NoSuchMethodException: Checked
- ArithmeticException: Unchecked
The throw Statement

- Exceptions are thrown using the *throw* statement
- Usually a throw statement is executed inside an if statement that evaluates a condition to see if the exception should be thrown
- See *CreatingExceptions.java*
- See *OutOfRangeException.java*
import java.util.Scanner;

public class CreatingExceptions
{
    //  Creates an exception object and possibly throws it.
    public static void main (String[] args) throws OutOfRangeException
    {
        final int MIN = 25, MAX = 40;

        Scanner scan = new Scanner (System.in);

        OutOfRangeException problem =
            new OutOfRangeException ("Input value is out of range.");

        continue
System.out.print("Enter an integer value between " + MIN + " and " + MAX + ", inclusive: ");
int value = scan.nextInt();

// Determine if the exception should be thrown
if (value < MIN || value > MAX)
    throw problem;

System.out.println("End of main method.");  // may never reach
Enter an integer value between 25 and 40, inclusive: 69
Exception in thread "main" OutOfRangeException:
  Input value is out of range.
  at CreatingExceptions.main(CreatingExceptions.java:20)

if (value < MIN || value > MAX)
    throw problem;

System.out.println("End of main method.");  // may never reach
public class OutOfRangeException extends Exception {
    public OutOfRangeException (String message) {
        super (message);
    }
}
What is the matter with this code?

```java
System.out.println("Before throw");
throw new OutOfRangeException("Too High");
System.out.println("After throw");
```
Quick Check

What is the matter with this code?

```java
System.out.println("Before throw");
throw new OutOfRangeException("Too High");
System.out.println("After throw");
```

The throw is not conditional and therefore always occurs. The second `println` statement can never be reached.
I/O Exceptions

• Let's examine issues related to exceptions and I/O

• A stream is a sequence of bytes that flow from a source to a destination

• In a program, we read information from an input stream and write information to an output stream

• A program can manage multiple streams simultaneously
Standard I/O

• There are three standard I/O streams:
  – *standard output* – defined by `System.out`
  – *standard input* – defined by `System.in`
  – *standard error* – defined by `System.err`

• We use `System.out` *when we execute* `println` *statements*

• `System.out` *and* `System.err` *typically represent* the console window

• `System.in` *typically represents* keyboard input, *which we've used many times with* `Scanner`
The IOException Class

• Operations performed by some I/O classes may throw an IOException
  – A file might not exist
  – Even if the file exists, a program may not be able to find it
  – The file might not contain the kind of data we expect

• An IOException is a checked exception
Writing Text Files

• In Chapter 5 we explored the use of the `Scanner` class to read input from a text file.

• Let's now examine other classes that let us write data to a text file.

• The `FileWriter` class represents a text output file, but with minimal support for manipulating data.

• Therefore, we also rely on `PrintStream` objects, which have `print` and `println` methods defined for them.
Writing Text Files

- Finally, we'll also use the `PrintWriter` class for advanced internationalization and error checking.
- We build the class that represents the output file by combining these classes appropriately.
- Output streams should be closed explicitly.
- See `TestData.java`
import java.util.Random;
import java.io.*;

public class TestData {
    //---
    // Creates a file of test data that consists of ten lines each
    // containing ten integer values in the range 10 to 99.
    //---
    public static void main (String[] args) throws IOException {
        final int MAX = 10;

        int value;
        String file = "test.dat";

        Random rand = new Random();
        continue
FileWriter fw = new FileWriter (file);
BufferedWriter bw = new BufferedWriter (fw);
PrintWriter outFile = new PrintWriter (bw);

for (int line=1; line <= MAX; line++)
{
    for (int num=1; num <= MAX; num++)
    {
        value = rand.nextInt (90) + 10;
        outFile.print (value + "   ");
    }
    outFile.println ();
}
outFile.close();
System.out.println ("Output file has been created: " + file);
FileWriter fw = new FileWriter(file);
BufferedWriter bw = new BufferedWriter(fw);
PrintWriter outFile = new PrintWriter(bw);

for (int line=1; line <= MAX; line++)
{
    for (int num=1; num <= MAX; num++)
    {
        value = rand.nextInt (90) + 10;
        outFile.print (value + "   ");
    }
    outFile.println();
}
outFile.close();
System.out.println ("Output file has been created: test.dat");

Output file has been created: test.dat

Sample test.dat File

77 46 24 67 45 37 32 40 39 10
90 91 71 64 82 80 68 18 83 89
25 80 45 75 74 40 15 90 79 59
44 43 95 85 93 61 15 20 52 86
60 85 18 73 56 41 35 67 21 42
93 25 89 47 13 27 51 94 76 13
33 25 48 42 27 24 88 18 32 17
71 10 90 88 60 19 89 54 21 92
45 26 47 68 55 98 34 38 98 38
48 59 90 12 86 36 11 65 41 62
Tool Tips

• A *tool tip* provides a short pop-up description when the mouse cursor rests momentarily on a component

• A tool tip is assigned using the `setToolTipText` method of a Swing component

```java
JButton button = new JButton("Compute");
button.setToolTipText("Calculate size");
```
Mnemonics

• A mnemonic is a keyboard alternative for pushing a button or selecting a menu option

• The mnemonic character should be chosen from the component's label, and is underlined

• The user activates the component by holding down the ALT+ctrl keys and pressing the mnemonic character

```java
JButton button = new JButton("Calculate");
button.setMnemonic("C");
```
Disabled Components

• Components can be disabled if they should not be used

• A disabled component is "grayed out" and will not respond to user interaction

• The status is set using the `setEnabled` method:

```java
JButton button = new JButton ("Do It");
button.setEnabled (false);
```
Light Bulb Example

• The right combination of special features such as tool tips and mnemonics can enhance the usefulness of a GUI

• See LightBulb.java
• See LightBulbPanel.java
• See LightBulbControls.java
import javax.swing.*;
import java.awt.*;

public class LightBulb
{
    // Sets up a frame that displays a light bulb image that can be
    // turned on and off.

    public static void main (String[] args)
    {
        JFrame frame = new JFrame ("Light Bulb");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        LightBulbPanel bulb = new LightBulbPanel();
        LightBulbControls controls = new LightBulbControls (bulb);

        continue
    }
}
continue

    JPanel panel = new JPanel();
    panel.setBackground(Color.black);
    panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
    panel.add(Box.createRigidArea(new Dimension(0, 20)));
    panel.add(bulb);
    panel.add(Box.createRigidArea(new Dimension(0, 10)));
    panel.add(controls);
    panel.add(Box.createRigidArea(new Dimension(0, 10)));

    frame.getContentPane().add(panel);
    frame.pack();
    frame.setVisible(true);
}
JPanel panel = new JPanel();
panel.setBackground(Color.black);
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
panel.add(Box.createRigidArea(new Dimension(0, 20)));
panel.add(bulb);
panel.add(Box.createRigidArea(new Dimension(0, 10)));
panel.add(controls);
panel.add(Box.createRigidArea(new Dimension(0, 10)));
frame.getContentPane().add(panel);
frame.pack();
frame.setVisible(true);
import javax.swing.*;
import java.awt.*;

public class LightBulbPanel extends JPanel {
    private boolean on;
    private ImageIcon lightOn, lightOff;
    private JLabel imageLabel;

    // continue
public LightBulbPanel()
{
    lightOn = new ImageIcon ("lightBulbOn.gif");
    lightOff = new ImageIcon ("lightBulbOff.gif");

    setBackground (Color.black);

    on = true;
    imageLabel = new JLabel (lightOff);
    add (imageLabel);
}
// Paints the panel using the appropriate image.
public void paintComponent (Graphics page)
{
    super.paintComponent(page);

    if (on)
        imageLabel.setIcon (lightOn);
    else
        imageLabel.setIcon (lightOff);
}

// Sets the status of the light bulb.
public void setOn (boolean lightBulbOn)
{
    on = lightBulbOn;
}

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class LightBulbControls extends JPanel
{
    private LightBulbPanel bulb;
    private JButton onButton, offButton;

    continue
// Sets up the lightbulb control panel.

public LightBulbControls (LightBulbPanel bulbPanel)
{
    bulb = bulbPanel;

    onButton = new JButton ("On");
onButton.setEnabled (false);
onButton.setMnemonic (n);
onButton.setToolTipText ("Turn it on!");
onButton.addActionListener (new OnListener());

    offButton = new JButton ("Off");
offButton.setEnabled (true);
offButton.setMnemonic (f);
offButton.setToolTipText ("Turn it off!");
offButton.addActionListener (new OffListener());

    setBackground (Color.black);
    add (onButton);
    add (offButton);
}
private class OnListener implements ActionListener {
    //--------------------------------------------------------------------------
    //  Turns the bulb on and repaints the bulb panel.
    //--------------------------------------------------------------------------
    public void actionPerformed (ActionEvent event)
    {
        bulb.setOn (true);
        onButton.setEnabled (false);
        offButton.setEnabled (true);
        bulb.repaint();
    }
}

continue
private class OffListener implements ActionListener {

    //------------------------------------------------------------------------------
    //  Turns the bulb off and repaints the bulb panel.
    //------------------------------------------------------------------------------
    public void actionPerformed(ActionEvent event) {
        bulb.setOn(false);
        onButton.setEnabled(true);
        offButton.setEnabled(false);
        bulb.repaint();
    }

}
Outline

Exception Handling
The try-catch Statement
Exception Classes
I/O Exceptions
Tool Tips and Mnemonics
Combo Boxes
Scroll Panes and Split Panes
Combo Boxes

• A *combo box* provides a menu from which the user can choose one of several options

• The currently selected option is shown in the combo box

• A combo box shows its options only when the user presses it using the mouse

• Options can be established using an array of strings or using the `addItem` method
The JukeBox Program

• A combo box generates an action event when the user makes a selection from it

• See JukeBox.java
• See JukeBoxControls.java
import javax.swing.*;

public class JukeBox
{
    // Creates and displays the controls for a juke box.
    public static void main (String[] args)
    {
        JFrame frame = new JFrame ("Java Juke Box");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

        JukeBoxControls controlPanel = new JukeBoxControls();

        frame.getContentPane().add(controlPanel);
        frame.pack();
        frame.setVisible(true);
    }
}
import javax.swing.*;

public class JukeBox {
    // --------------------------------------------------------------------------------------------------
    // Creates and displays the controls for a juke box.
    // --------------------------------------------------------------------------------------------------
    public static void main (String[] args) {
        JFrame frame = new JFrame("Java Juke Box");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JukeBoxControls controlPanel = new JukeBoxControls();

        frame.getContentPane().add(controlPanel);
        frame.pack();
        frame.setVisible(true);
    }
}
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.applet.AudioClip;
import java.net.URL;

public class JukeBoxControls extends JPanel {
    private JComboBox musicCombo;
    private JButton stopButton, playButton;
    private AudioClip[] music;
    private AudioClip current;

    // Sets up the GUI for the juke box.
    public JukeBoxControls() {
        URL url1, url2, url3, url4, url5, url6;
        url1 = url2 = url3 = url4 = url5 = url6 = null;
    }
}

// Obtain and store the audio clips to play
try {
    url1 = new URL("file", "localhost", "westernBeat.wav");
    url2 = new URL("file", "localhost", "classical.wav");
    url3 = new URL("file", "localhost", "jeopardy.au");
    url4 = new URL("file", "localhost", "newAgeRythm.wav");
    url5 = new URL("file", "localhost", "eightiesJam.wav");
    url6 = new URL("file", "localhost", "hitchcock.wav");
} catch (Exception exception) {}

music = new AudioClip[7];
music[0] = null; // Corresponds to "Make a Selection"
music[1] = JApplet.newAudioClip (url1);
music[2] = JApplet.newAudioClip (url2);
music[3] = JApplet.newAudioClip (url3);
music[4] = JApplet.newAudioClip (url4);
music[5] = JApplet.newAudioClip (url5);
music[6] = JApplet.newAudioClip (url6);

JLabel titleLabel = new JLabel("Java Juke Box");
titleLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
// Create the list of strings for the combo box options
String[] musicNames = {
    "Make A Selection...", "Western Beat",
    "Classical Melody", "Jeopardy Theme", "New Age Rythm",
    "Eighties Jam", "Alfred Hitchcock's Theme"};
musicCombo = new JComboBox (musicNames);
musicCombo.setAlignmentX (Component.CENTER_ALIGNMENT);

// Set up the buttons
playButton = new JButton ("Play", new ImageIcon ("play.gif");
playButton.setBackground (Color.white);
playButton.setMnemonic ('p');
stopButton = new JButton ("Stop", new ImageIcon ("stop.gif");
stopButton.setBackground (Color.white);
stopButton.setMnemonic ('s');

JPanel buttons = new JPanel();
buttons.setLayout (new BoxLayout (buttons, BoxLayout.X_AXIS));
buttons.add (playButton);
buttons.add (Box.createRigidArea (new Dimension(5,0)));
buttons.add (stopButton);
buttons.setBackground (Color.cyan);

// Set up this panel
setPreferredSize (new Dimension (300, 100));
setBackground (Color.cyan);
setLayout (new BoxLayout (this, BoxLayout.Y_AXIS));
continue

musicCombo.addActionListener (new ComboListener());
stopButton.addActionListener (new ButtonListener());
playButton.addActionListener (new ButtonListener());

current = null;
}

//*******************************************************************
// Represents the action listener for the combo box.
//*******************************************************************
private class ComboListener implements ActionListener
{
    // --------------------------------------------------------------
    // Stops playing the current selection (if any) and resets
    // the current selection to the one chosen.
    // --------------------------------------------------------------
    public void actionPerformed (ActionEvent event)
    {
        if (current != null)
            current.stop();

        current = music[musicCombo.getSelectedIndex()];
    }
}

continue
continue

//****************************************************************************
//  Represents the action listener for both control buttons.
//*****************************************************************************
private class ButtonListener implements ActionListener
{
    //-----------------------------------------------
    //   Stops the current selection (if any) in either case. If
    //   the play button was pressed, start playing it again.
    //-----------------------------------------------
    public void actionPerformed (ActionEvent event)
    {
        if (current != null)
            current.stop();

        if (event.getSource() == playButton)
            if (current != null)
                current.play();
    }
}
Outline

Exception Handling
The try-catch Statement
Exception Classes
I/O Exceptions
Tool Tips and Mnemonics
Combo Boxes
Scroll Panes and Split Panes
Scroll Panes

• A *scroll pane* is useful for images or information too large to fit in a reasonably-sized area

• A scroll pane offers a limited view of the component it contains

• It provides vertical and/or horizontal scroll bars that allow the user to scroll to other areas of the component

• No event listener is needed for a scroll pane

• See TransitMap.java
import java.awt.*;
import javax.swing.*;

public class TransitMap
{
    public static void main (String[] args)
    {
        // SEPTA = SouthEast Pennsylvania Transit Authority
        JFrame frame = new JFrame ("SEPTA Transit Map");
    }
}
continue

frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);

ImageIcon image = new ImageIcon ("septa.jpg");
JLabel imageLabel = new JLabel (image);

JScrollPane sp = new JScrollPane (imageLabel);
sp.setPreferredSize (new Dimension (450, 400));

frame.getContentPane().add (sp);
frame.pack();
frame.setVisible(true);
}
frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
ImageIcon image = new ImageIcon ("septa.jpg");
JLabel imageLabel = new JLabel (image);
JScrollPane sp = new JScrollPane (imageLabel);
sp.setPreferredSize (new Dimension (450, 400));
frame.getContentPane().add (sp);
frame.pack();
frame.setVisible(true);
Split Panes

• A split pane is a container that displays two components separated by a moveable divider bar.

• The two components can be displayed side by side, or one on top of the other.
Split Panes

• The orientation of the split pane is set using the HORIZONTAL_SPLIT or VERTICAL_SPLIT constants

• The divider bar can be set so that it can be fully expanded with one click of the mouse

• The components can be continuously adjusted as the divider bar is moved, or wait until it stops moving

• Split panes can be nested
Lists

• The Swing `JList` class represents a list of items from which the user can choose

• The contents of a `JList` object can be specified using an array of objects

• A `JList` object generates a `list selection event` when the current selection changes

• See `PickImage.java`

• See `ListPanel.java`
//********************************************************************
//  PickImage.java       Authors: Lewis/Loftus
//
//  Demonstrates the use a split pane and a list.
//********************************************************************

import java.awt.*;
import javax.swing.*;

public class PickImage
{
    //-----------------------------------------------------------------
    //  Creates and displays a frame containing a split pane. The
    //  user selects an image name from the list to be displayed.
    //-----------------------------------------------------------------
    public static void main (String[] args)
    {
        JFrame frame = new JFrame ("Pick Image");
        frame.setDefaultCloseOperation (JFrame.EXIT_ON_CLOSE);
    
    
    
    continue
continue

JLabel imageLabel = new JLabel();
JPanel imagePanel = new JPanel();
imagePanel.add (imageLabel);
imagePanel.setBackground (Color.white);

ListPanel imageList = new ListPanel (imageLabel);
// imageList has a reference to the JLabel shown on
// imagePanel – it can change the Icon shown on
// imagePanel
JSplitPane sp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
    imageList, imagePanel);

sp.setOneTouchExpandable (true);

frame.getContentPane().add (sp);
frame.pack();
frame.setVisible(true);
}
JLabel imageLabel = new JLabel();
JPanel imagePanel = new JPanel();
imagePanel.add(imageLabel);
imagePanel.setBackground(Color.white);
ListPanel imageList = new ListPanel(imageLabel);
JSplitPane sp = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT,
imageList, imagePanel);
sp.setOneTouchExpandable(true);
frame.getContentPane().add(sp);
frame.pack();
frame.setVisible(true);
//********************************************************************
// ListPanel.java       Authors: Lewis/Loftus
//
// Represents the list of images for the PickImage program.
//********************************************************************

import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

public class ListPanel extends JPanel {
    private JLabel label;
    private JList list;
}

continue
public ListPanel (JLabel imageLabel)
{
    label = imageLabel;

    String[] fileNames = { "circuit.gif",
                          "duke.gif",
                          "hammock.gif",
                          "justin.jpg",
                          "kayla.jpg",
                          "tiger.jpg",
                          "toucan.gif",
                          "worldmap.gif" }; 

    list = new JList (fileNames);
    list.addListSelectionListener (new ListListener());
    list.setSelectionMode (ListSelectionModel.SINGLE_SELECTION);

    add (list);
    setBackground (Color.white);
}
private class ListListener implements ListSelectionListener {
    public void valueChanged (ListSelectionEvent event) {
        if (list.isSelectionEmpty())
            label.setIcon (null);
        else {
            String fileName = (String)list.getSelectedValue();
            ImageIcon image = new ImageIcon (fileName);
            label.setIcon (image);
        }
    }
}
Lists

• A `JList` object can be set so that multiple items can be selected at the same time

• The *list selection mode* can be one of three options:
  – single selection – only one item can be selected at a time
  – single interval selection – multiple, contiguous items can be selected at a time
  – multiple interval selection – any combination of items can be selected

• The list selection mode is defined by a `ListSelectionModel` object
Summary

• Chapter 11 has focused on:
  • the purpose of exceptions
  • exception messages
  • the try-catch statement
  • propagating exceptions
  • the exception class hierarchy
  • GUI mnemonics and tool tips
  • more GUI components and containers