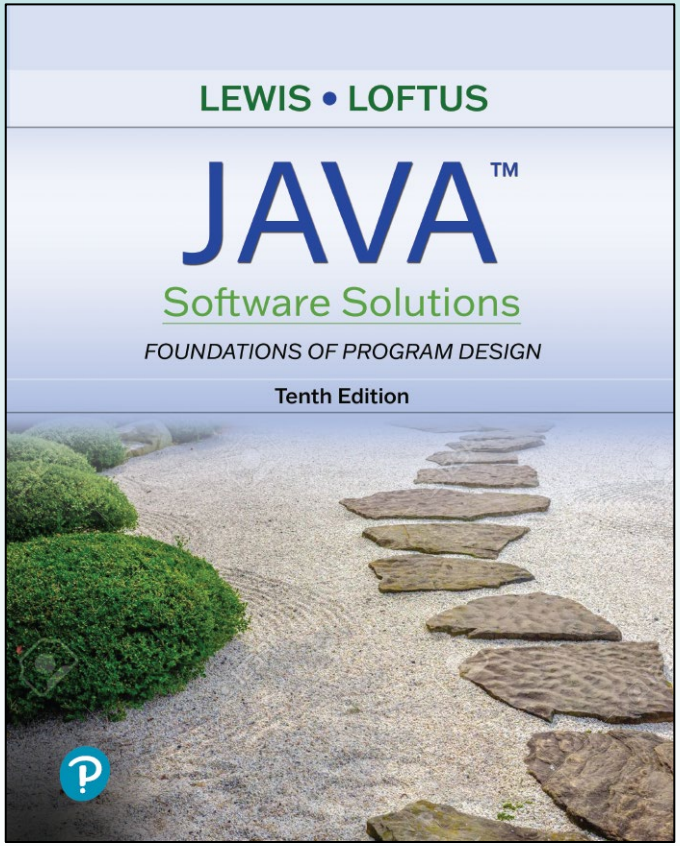
Chapter 11
Exceptions



Java Software Solutions

Foundations of Program Design

10th Edition

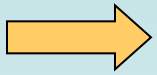
(edited BPK 3/23/2017, 03/24/2019, 08/12/2026)

John Lewis
William Loftus

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
  - I/O exceptions and writing text files
  - GUI tool tips and disabled controls
  - more GUI controls

Outline



Exception Handling

The try-catch Statement

Exception Classes

I/O Exceptions

Tool Tips and Disabling Controls

Scroll Panes

Split Panes and List Views

Exceptions

- An *Exception* is an object that describes an unusual or erroneous situation
- Exceptions are *thrown* by a program when some sort of problem occurs, 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

- Java has predefined exceptions that can occur during execution
- The programmer can define additional exceptions
- Some exceptions are due to outright errors and exception handling provides a way to detect them and recover.
- Other exceptions are due to rare situations and exception handling provides a way to process them outside of the normal flow.

Exception Handling

- To deal with an exception:
  - ignore it
  - handle it where it occurs
  - handle it an another place in the program
- Which of these to use is a design consideration
- Commonly occurring situations should be handled with the usual flow-control logic

Exception Handling: ignore it

- If an exception is ignored (not caught) by the program, the program will terminate and produce an error message
- The message includes a *call stack trace* that:
  - shows the problem
  - indicates the line on which the exception occurred
  - shows the trail that lead to the problem
- See `Zero.java`

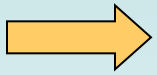
```
//********************************************************************
//  Zero.java       Author: Lewis/Loftus
//
//  Demonstrates an uncaught exception.
//********************************************************************

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.");
   }
}
```

Output (when program terminates)

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Zero.main(Zero.java:17)
```

```
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

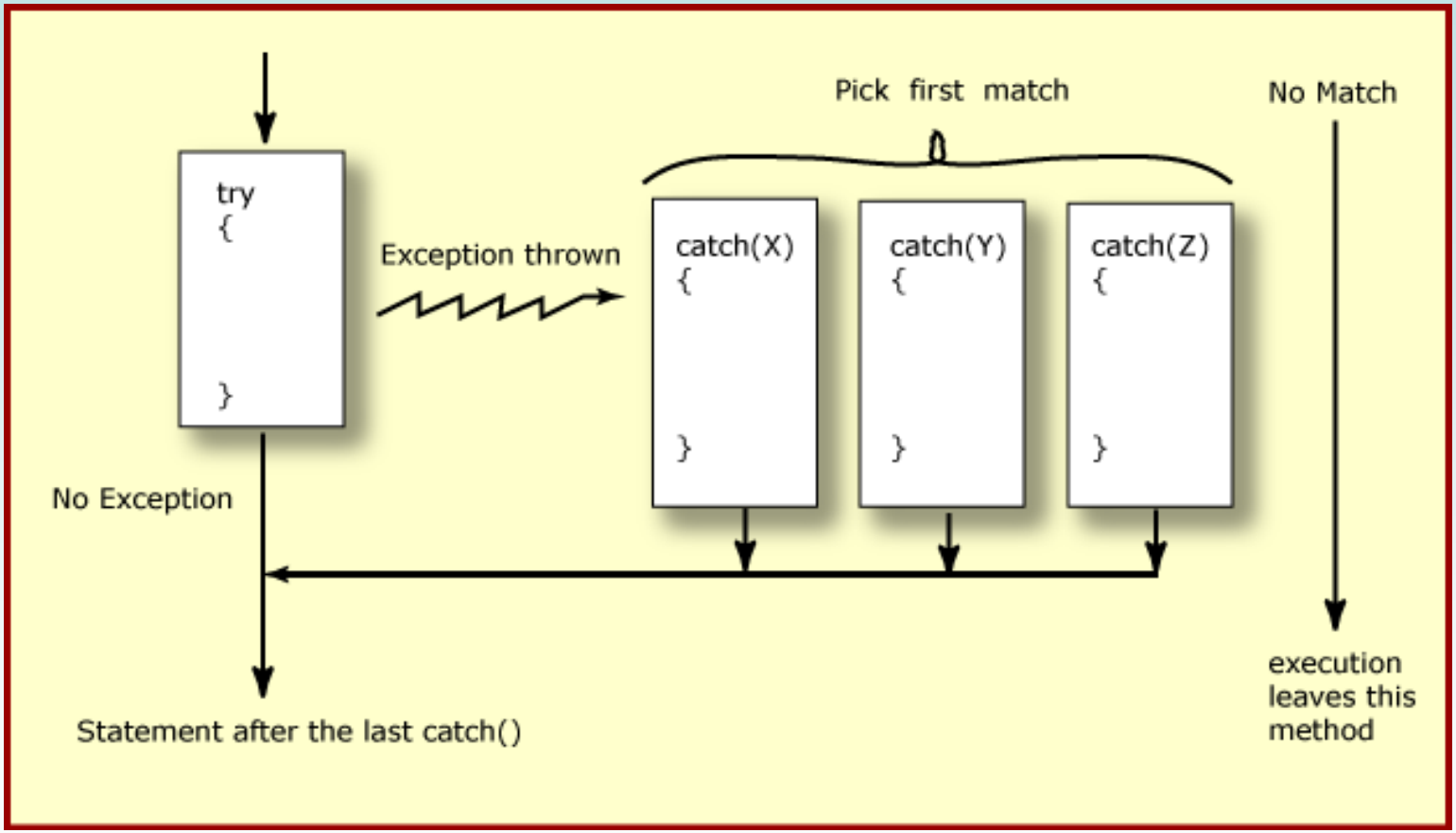
Tool Tips and Mnemonics

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
- A exception matches a catch clause if the clause lists that exception type or an ancestor of it.



Statements throw Exceptions

- Division by zero throws an `ArithmeticException`
- Using an out-of-bounds index with an array throws an `ArrayIndexOutOfBoundsException`
- Using an out-of-bounds index with a String throws a `StringIndexOutOfBoundsException`
- Dozens more

Methods throw Exceptions

- Some methods throw an exception when they encounter a problem
- Methods that use these methods may deal with these exceptions

```
public static int parseInt(String s) throws NumberFormatException
```

- See `ProductCodes.java`
  - a company may not sell products from zone R in districts with code >= 2000

```
//********************************************************************
//  ProductCodes.java       Author: Lewis/Loftus
//
//  TRV2475A5R-14      character  at index 9          == zone
//  0123456789012      characters at indexes 3,4,5,6 == district
//********************************************************************

import java.util.Scanner;

public class ProductCodes
{
   //-----------------------------------------------------------------
   //  Counts the number of product codes that are entered with a
   //  zone of R and a district greater than 2000.
   //-----------------------------------------------------------------
   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();
```

```
      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);
   }
}
```

```
        while (!code.equals ("XXX"))
        {
            try
            {
                zone = code.charAt(9);
                district = Integer.parseInt(code.substring(3, 7));
                valid++;
                if (zone == 'R' && district > 2000)
                    banned++;
            }
            catch (Strin
            {
                System.ou
            }
            catch (Numbe
            {
                System.ou
            }

            System.out.p
            code = scan.
        }

        System.out.prin
        System.out.prin
    }
}
```

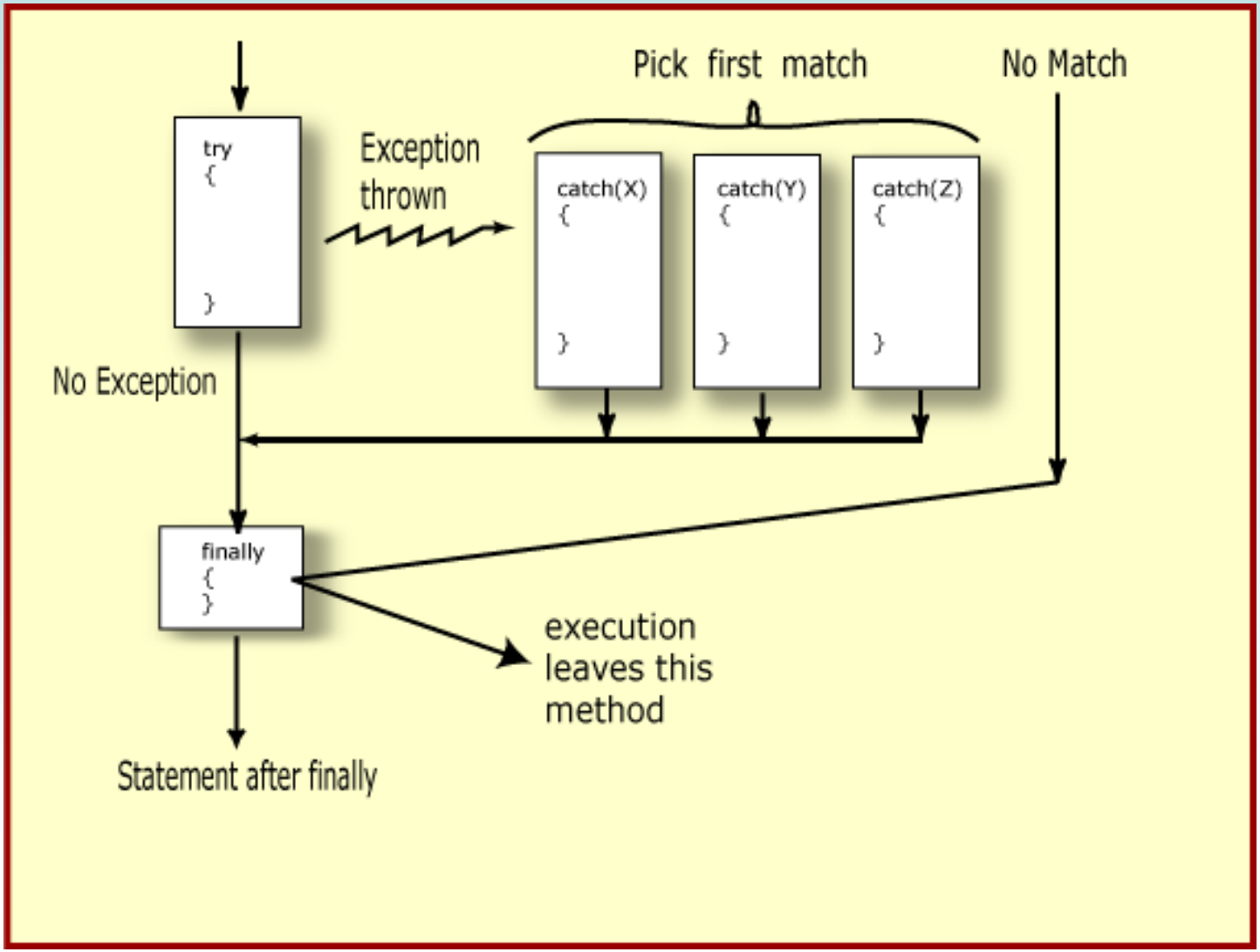
Sample Run

```
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
```

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

finally



Exception Propagation

- An exception is handled at a higher level if it is not handled 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`

```
//********************************************************************
//  Propagation.java       Author: Lewis/Loftus
//
//  Demonstrates exception propagation.
//********************************************************************

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

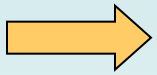
```
Program beginning.
Level 1 beginning.
Level 2 beginning.
Level 3 beginning.

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.
```

```
//********************************************************************
//  ExceptionScope.java       Author: Lewis/Loftus
//
//  Demonstrates exception propagation.
//********************************************************************

public class ExceptionScope
{
   //-----------------------------------------------------------------
   //  Catches and handles the exception that is thrown in level3.
   //-----------------------------------------------------------------
   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();
```

```
            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.");
    }
```

```
   //-----------------------------------------------------------------
   //  Performs a calculation to produce an exception.  It is not
   //  caught and handled at this level.
   //-----------------------------------------------------------------
   public void level3()
   {
      int numerator = 10, denominator = 0;

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

Outline

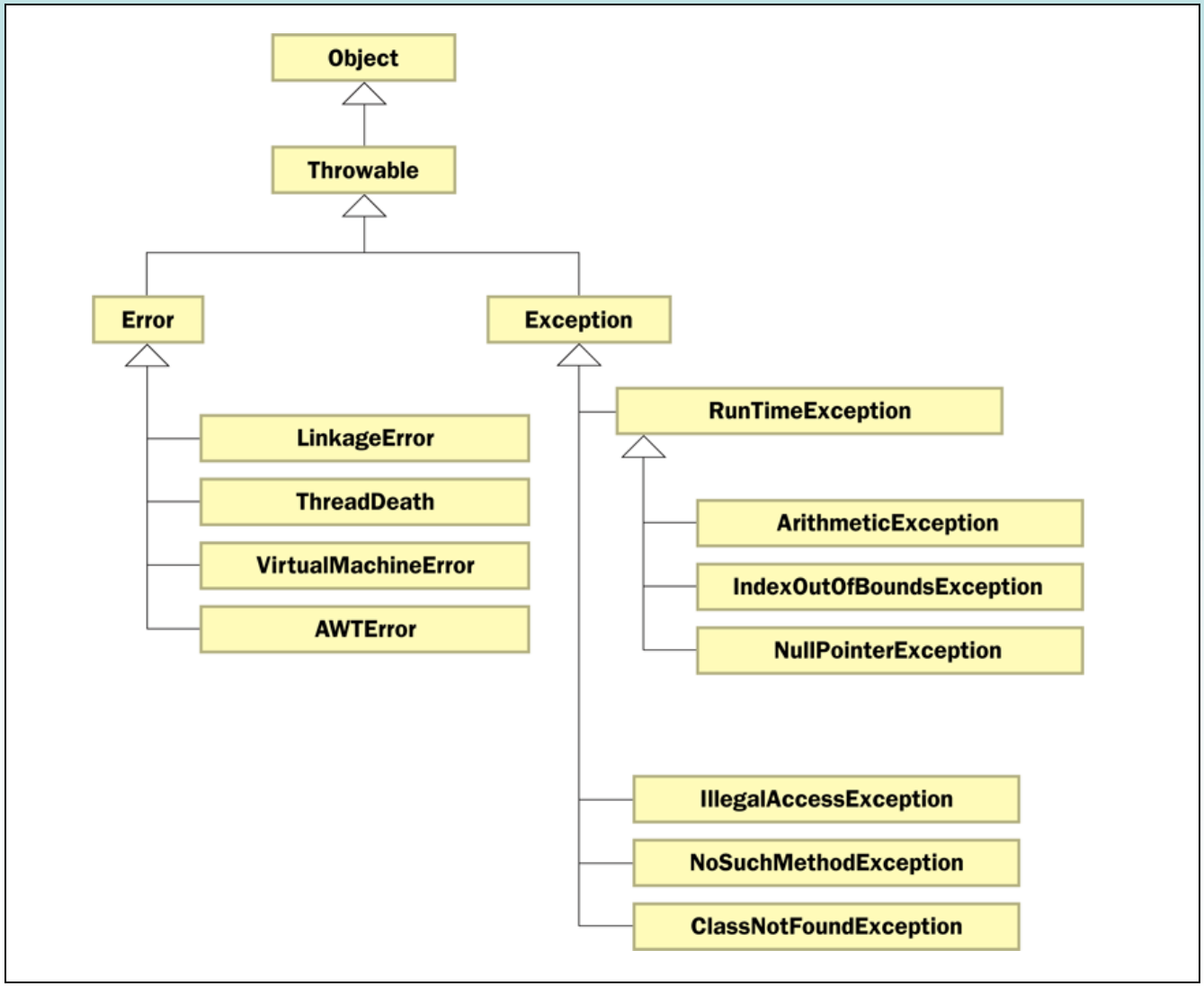


- Exception Handling
- The try-catch Statement
- Exception Classes
- I/O Exceptions
  - ☺ Tool Tips and Mnemonics
  - ☺ Combo Boxes
  - ☺ Scroll Panes and Split Panes

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 Exception Class Hierarchy



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`

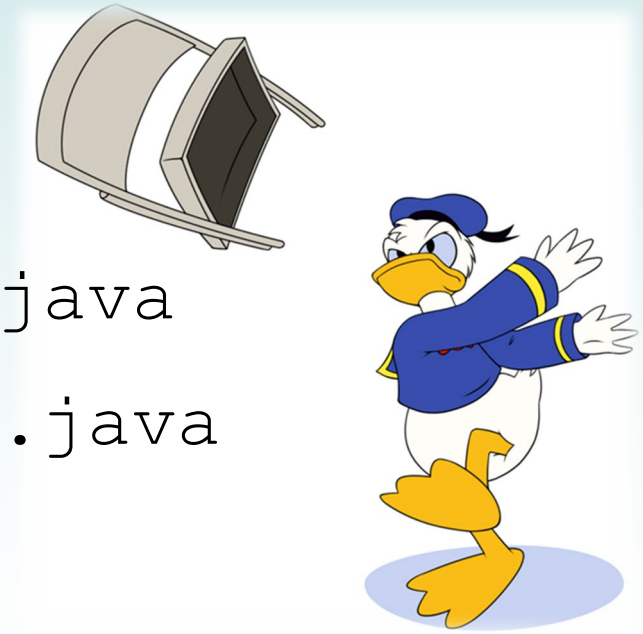
Quick Check

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`

```
//********************************************************************
//  CreatingExceptions.java       Author: Lewis/Loftus
//
//  Demonstrates the ability to define an exception via inheritance.
//********************************************************************

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

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
   }
}
```

Sample Run

```
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
    }
}
```

```
//********************************************************************
//  OutOfRangeException.java       Author: Lewis/Loftus
//
//  Represents an exceptional condition in which a value is out of
//  some particular range.
//********************************************************************

public class OutOfRangeException extends Exception
{
   //-----------------------------------------------------------------
   //  Sets up the exception object with a particular message.
   //-----------------------------------------------------------------
   OutOfRangeException(String message)
   {
      super(message);
   }
}
```

Quick Check

What is the matter with this code?

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

Quick Check

What is the matter with this code?

```
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.

Outline

Exception Handling

The try-catch Statement

Exception Classes

I/O Exceptions

Tool Tips and Disabling Controls

Scroll Panes

Split Panes and List Views

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
- An actual I/O device is complicated and hard to manage
  - A stream makes it easy to use
- 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
- Chapter 5 used the `Scanner` class to read input from a text file

```
import java.util.Scanner;
import java.io.*;

public class URLDissector  // from chapter 5
{
   public static void main(String[] args) throws IOException
   {
      String url;
      Scanner fileScan, urlScan;
      fileScan = new Scanner(new File("urls.inp"));

      // Read and process each line of the file
      while ( fileScan.hasNext() )
      {
         url = fileScan.nextLine();
         System.out.println("URL: " + url);

         // set up a Scanner for this line
         urlScan = new Scanner(url);
         urlScan.useDelimiter("/");     // tokens are separated by /

         //  Print each part of the url
         while ( urlScan.hasNext() )
            System.out.println ( "    " + urlScan.next() );

         System.out.println();
      }
   }
}
```

Writing Text Files

- The **`PrintWriter`** class represents a text output file
- The constructor **`PrintWriter`** for may throw a **`FileNotFoundException`**
- Its write methods do not throw exceptions
- Output streams should be closed explicitly with close()
- See `TestData.java`

```
//********************************************************************
//  TestData.java       Author: Lewis/Loftus
//
//  Demonstrates I/O exceptions and the use of a character file
//  output stream.
//********************************************************************

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 fileName = "test.txt";  // could get this from user

      PrintWriter outFile = new PrintWriter(fileName);
```

continue

```
        Random rand = new Random();

        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: " + fileName);
    }
}
```

Output

```
Output file has been created: test.txt
```

Sample test.txt 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
```

Outline

Exception Handling

The try-catch Statement

Exception Classes

I/O Exceptions

Tool Tips and Disabling Controls

Scroll Panes

Split Panes and List Views

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 `setToolTip` method of a JavaFX control

```
ToolTip tip = new ToolTip("Update cost");
myButton.setToolTip(tip);
```

Disabled Controls

- Controls can be *disabled* if they should not be used
- A disabled control is "grayed out" and will not respond to user interaction
- To disable a control, call the `setDisable` method:

```
myButton.setDisable(true);
```

- See `LightBulb.java`

```
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.geometry.Pos;
import javafx.geometry.Rectangle2D;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Tooltip;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

//*******************************************************************
//  LightBulb.java       Author: Lewis/Loftus
//
//  Demonstrates the use of tool tips and disabled controls.
//*******************************************************************

public class LightBulb extends Application
{
    private Button onButton, offButton;
    private ImageView bulbView;
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Displays an image of a light bulb that can be turned on and off
    //  using enabled buttons with tool tips set.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        Image img = new Image("lightBulbs.png");
        bulbView = new ImageView(img);
        bulbView.setViewport(new Rectangle2D(0, 0, 125, 200));  // off

        onButton = new Button("On");
        onButton.setPrefWidth(70);
        onButton.setTooltip(new Tooltip("Turn me on!"));
        onButton.setOnAction(this::processButtonPress);

        offButton = new Button("Off");
        offButton.setPrefWidth(70);
        offButton.setTooltip(new Tooltip("Turn me off!"));
        offButton.setDisable(true);
        offButton.setOnAction(this::processButtonPress);
```

continue

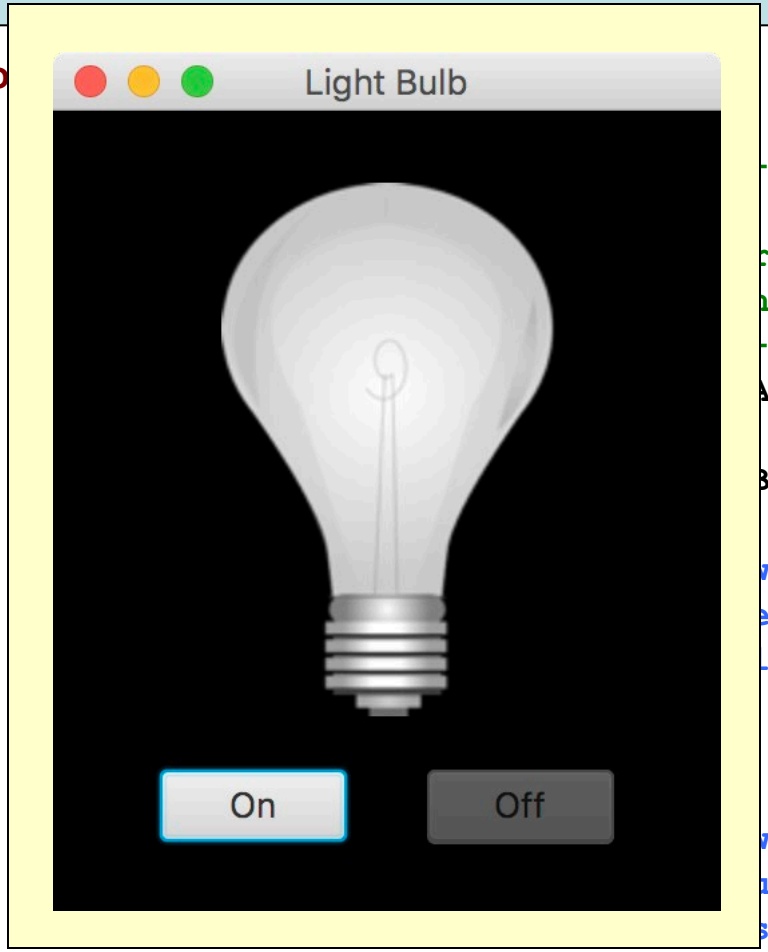
continue

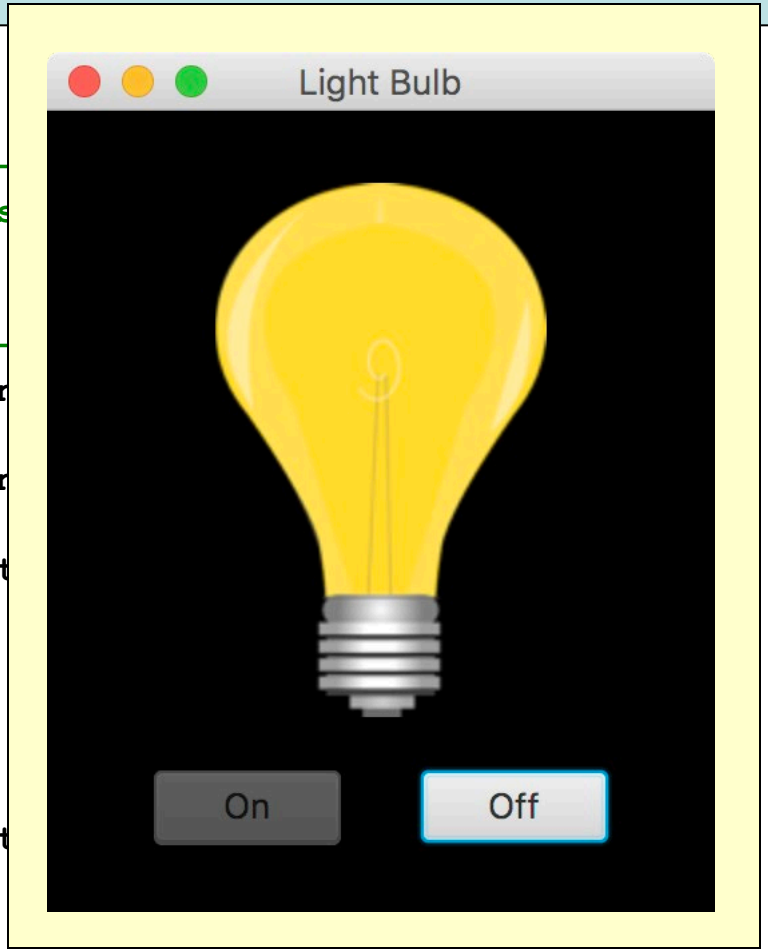
```
        HBox buttons = new HBox(onButton, offButton);
        buttons.setAlignment(Pos.CENTER);
        buttons.setSpacing(30);

        VBox root = new VBox(bulbView, buttons);
        root.setAlignment(Pos.CENTER);
        root.setStyle("-fx-background-color: black");
        root.setSpacing(20);

        Scene scene = new Scene(root, 250, 300);

        primaryStage.setTitle("Light Bulb");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Determines which button was pressed and sets the image viewport
    //  appropriately to show either the on or off bulb. Also swaps the
    //  disable state of both buttons.
    //-----------------------------------------------------------------
    public void processButtonPress(ActionEvent event)
    {
        if (event.getSource() == onButton)
        {
            bulbView.setViewport(new Rectangle2D(143, 0, 125, 200)); // on
            onButton.setDisable(true);
            offButton.setDisable(false);
        }
        else
        {
            bulbView.setViewport(new Rectangle2D(0, 0, 125, 200)); // off
            offButton.setDisable(true);
            onButton.setDisable(false);
        }
    }
}
```






```
co
                                    press
                                    the
                                    ns.
                                    Actior
                                    Buttor
                                    Rect
                                    e);
                                    lse);
                                    Rect
                                    ue);
                                    se);
        }
    }
}
```

Outline

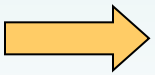
Exception Handling

The try-catch Statement

Exception Classes

I/O Exceptions

Tool Tips and Disabling Controls



Scroll Panes

Split Panes and List Views

Scroll Panes

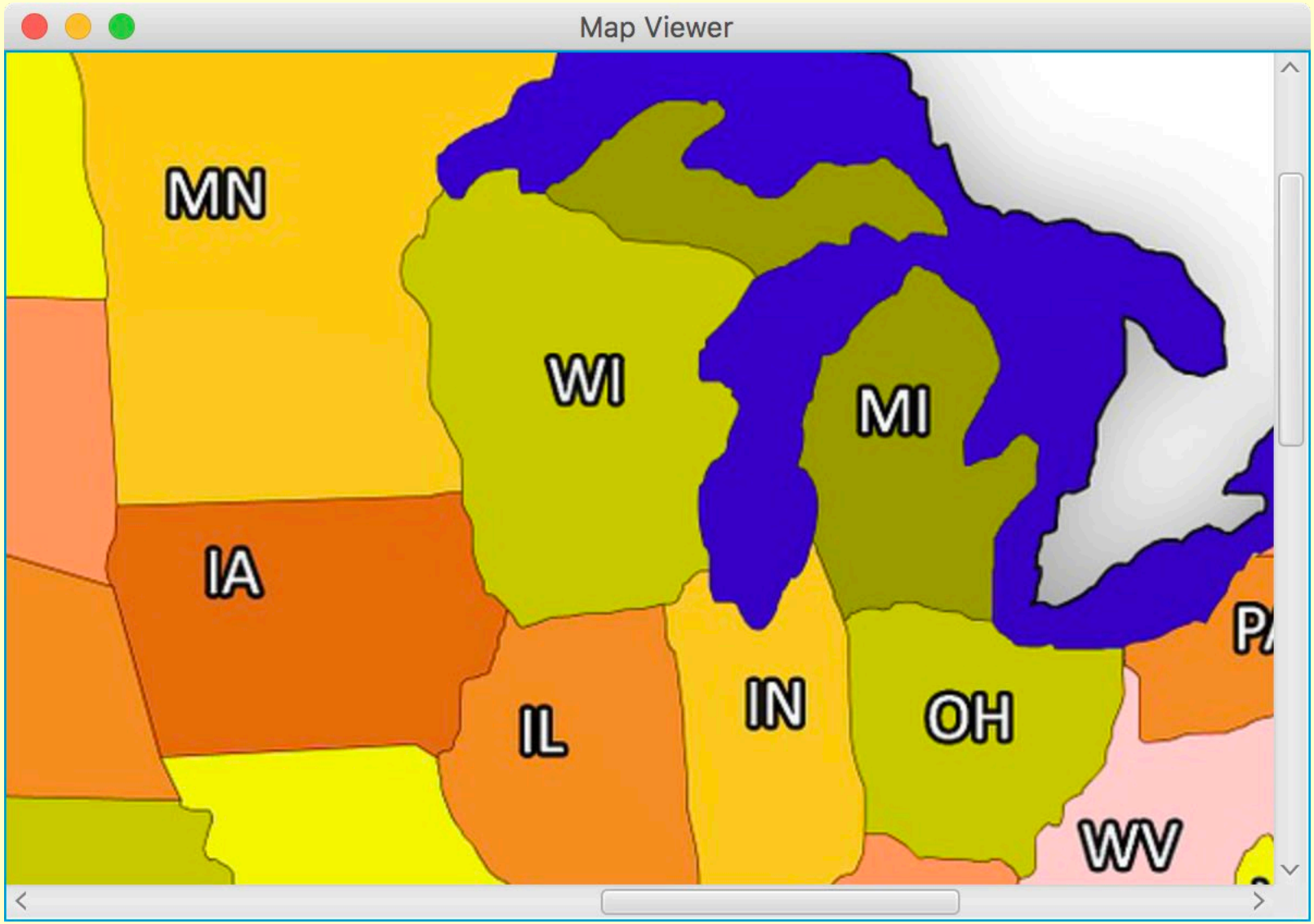
- A JavaFX *scroll pane* provides a limited but scrollable view of a large underlying node, such as an image
- A scroll pane provides horizontal and vertical scroll bars, as needed
- No event listener is needed for a scroll pane
- See `MapViewer.java`

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.ScrollPane;
import javafx.scene.control.ScrollPane.ScrollBarPolicy;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.stage.Stage;

//*****************************************************************
//  MapViewer.java       Author: Lewis/Loftus
//
//  Demonstrates the use of a scroll pane.
//*****************************************************************

public class MapViewer extends Application
{
    //-----------------------------------------------------------------
    //  Presents a scroll pane that allows the user to determine which
    //  section of the underlying image (a map of the USA) is visible.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
```

continue

continue

```
        Image img = new Image("map.jpg");
        ImageView imgView = new ImageView(img);

        ScrollPane root = new ScrollPane(imgView);
        root.setHbarPolicy(ScrollBarPolicy.ALWAYS);

        Scene scene = new Scene(root, 600, 400);

        primaryStage.setTitle("Map Viewer");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```



Outline

Exception Handling

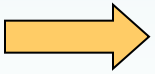
The try-catch Statement

Exception Classes

I/O Exceptions

Tool Tips and Disabling Controls

Scroll Panes



Split Panes and List Views

Split Panes

- A JavaFX *split pane* displays two or more nodes separated by a moveable divider bar
- As the user slides the divider bar, it gives more space to one side and reduces space on the other
- The nodes can be displayed side by side or one on top of the other
- If more than two nodes are added to a split pane, each node is separated from the next by another divider bar

List Views

- A JavaFX *list view* presents a list of selectable options
- The options are always visible (not a drop-down), and the view is resizable and scrollable
- A list view contains `Observable` items
- The following example displays a list of words (food items) on one side of a split pane and an image of the selected food item on the other
- See `FoodImages.java`

```
import javafx.application.Application;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.scene.Scene;
import javafx.scene.control.ListView;
import javafx.scene.control.SplitPane;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

//*******************************************************************
//  FoodImages.java       Author: Lewis/Loftus
//
//  Demonstrates a split pane and a list view.
//*******************************************************************

public class FoodImages extends Application
{
    private Image[] foodImages;
    private ImageView imgView;
    private ListView<String> listView;
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Displays a split pane with a list of food items on the left
    //  and an image of the selected food item on the right.
    //-----------------------------------------------------------------
    public void start(Stage primaryStage)
    {
        String[] food = {"apples", "asparagus", "bacon", "bread",
            "carrots", "cheesecake", "eggs", "hamburger", "muffins",
            "onions", "oranges", "pancakes", "peanuts", "pizza",
            "potatoes", "pretzels", "spaghetti", "sushi", "watermelon"};

        foodImages = new Image[food.length];
        for (int i = 0; i < food.length; i++)
            foodImages[i] = new Image(food[i] + ".jpg");

        imgView = new ImageView(foodImages[0]);
        StackPane imgPane = new StackPane(imgView);
        imgPane.setMinWidth(300);

        imgView.setPreserveRatio(true);
        imgView.fitWidthProperty().bind(imgPane.widthProperty());

        ObservableList<String> list = FXCollections.observableArrayList();
        list.addAll(food);
```

continue

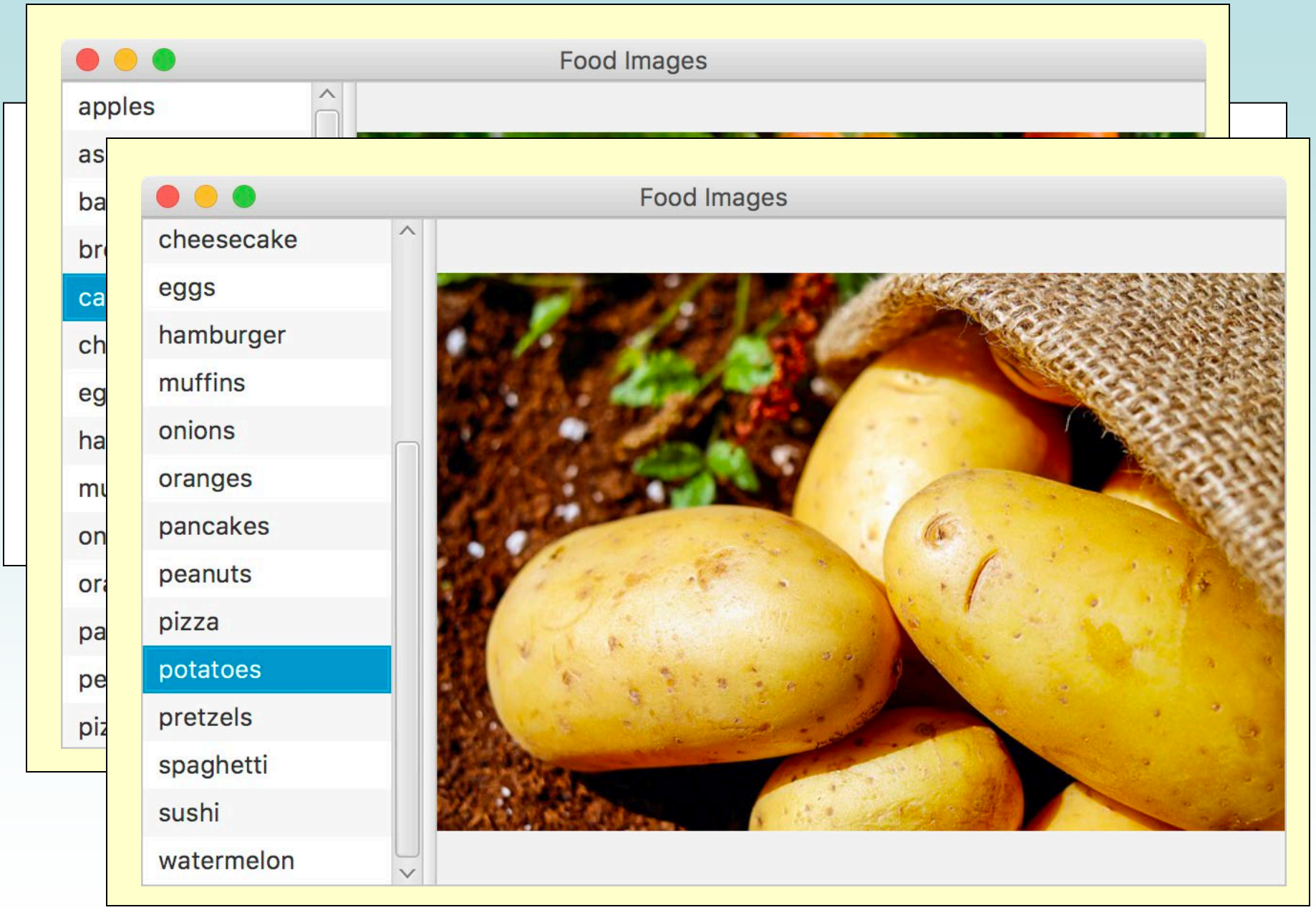
continue

```
        listView = new ListView<String>(list);
        listView.setMinWidth(100);
        listView.getSelectionModel().select(0);
        listView.getSelectionModel().selectedItemProperty().addListener(
            this::processListSelection);

        SplitPane root = new SplitPane();
        root.setDividerPositions(0.25);
        root.getItems().addAll(listView, imgPane);

        Scene scene = new Scene(root, 600, 350);

        primaryStage.setTitle("Food Images");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

continue

continue

```
    //-----------------------------------------------------------------
    //  Processes a list view selection by getting the index of the
    //  selected item and displaying the corresponding image.
    //-----------------------------------------------------------------
    public void processListSelection(ObservableValue<? extends String> val,
            String oldValue, String newValue)
    {
        int index = listView.getSelectionModel().getSelectedIndex();
        imgView.setImage(foodImages[index]);
    }
}
```



Summary

- Chapter 11 has focused on:
  - the purpose of exceptions
  - exception messages
  - the try-catch statement
  - propagating exceptions
  - I/O exceptions and writing text files
  - GUI tool tips and disabled controls
  - more GUI controls