CMPE 101
Object Oriented Programming

İstanbul
Bilgi University

Dr. Emel Küpçü

Department of Computer Engineering

İstanbul Bilgi University

Week-13: UML Diagrams & Graphical User Interfaces

UML Diagrams

- UML (Unified Modeling Language) diagrams are used in Java development to visually represent the structure and behavior of a system.

- They help developers, designers, and stakeholders understand, design, and communicate the architecture of software effectively.

Benefits of Using UML Diagrams in Java

- **Better Understanding of the System:**
  UML diagrams help visualize the components and their relationships, making it easier to understand complex systems.
- **Efficient Design and Planning:**
  They provide a common language for developers, analysts, and clients, enhancing communication and reducing misunderstandings.
- Before coding begins, UML diagrams allow developers to plan and design the system architecture, which can prevent design flaws early in the process.
- **Code Documentation:**
  UML diagrams act as documentation that describes the structure (class diagrams) and behavior (sequence, activity diagrams) of the application.
- **Support for Object-Oriented Design:**
  UML is especially useful in Java because it aligns well with object-oriented concepts like classes, inheritance, and interfaces.
- **Tool Integration:**
  Many IDEs (like IntelliJ IDEA, Eclipse) support UML diagrams, enabling automatic generation and synchronization with code, which saves time.

```
1  // Fig. 3.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account {
6     private String name; // instance variable
7
8     // method to set the name in the object
9     public void setName(String name) {
10       this.name = name; // store the name
11    }
12
13    // method to retrieve the name from the object
14    public String getName() {
15       return name; // return value of name to caller
16    }
17 }
```

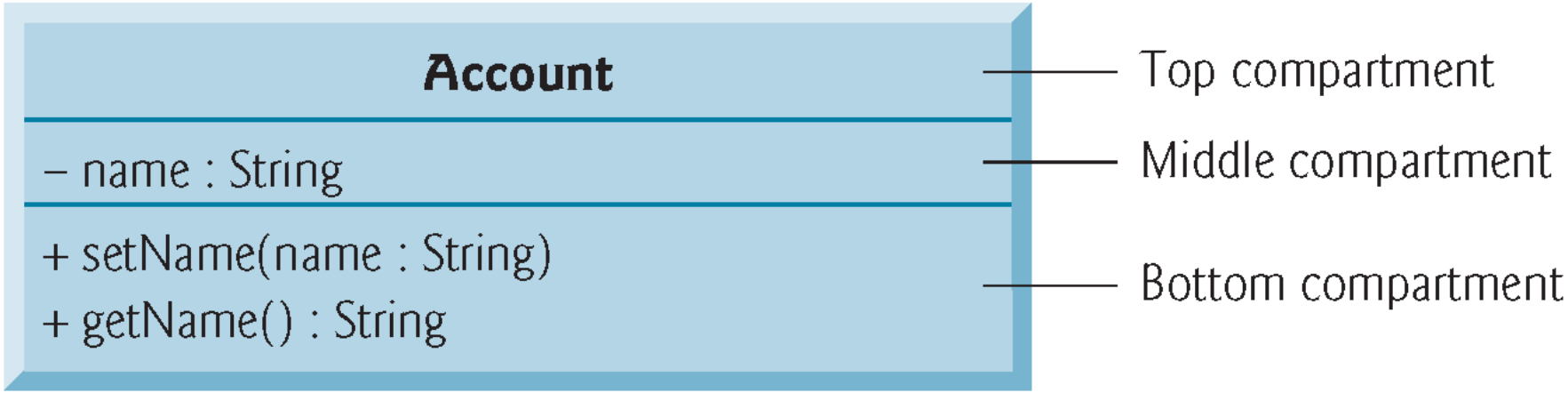
Fig. 3.1 | `Account` class that contains a `name` instance variable and methods to *set* and *get* its value.

3.2.4 Account UML Class Diagram

- In the UML, each class is modeled in a class diagram as a rectangle with three compartments.
- The **top compartment** contains the class's name centered horizontally in boldface.
- The **middle compartment** contains the class's attributes, which correspond to instance variables in Java.

Bottom Compartment:

- The bottom compartment contains the class's operations, which correspond to methods and constructors in Java.
- The UML represents instance variables as an attribute name, followed by a colon and the type.
- Private attributes are preceded by a minus sign (–) in the UML.
- The UML models operations by listing the operation name followed by a set of parentheses.
- A plus sign (+) in front of the operation name indicates that the operation is a public one in the UML (i.e., a `public` method in Java).



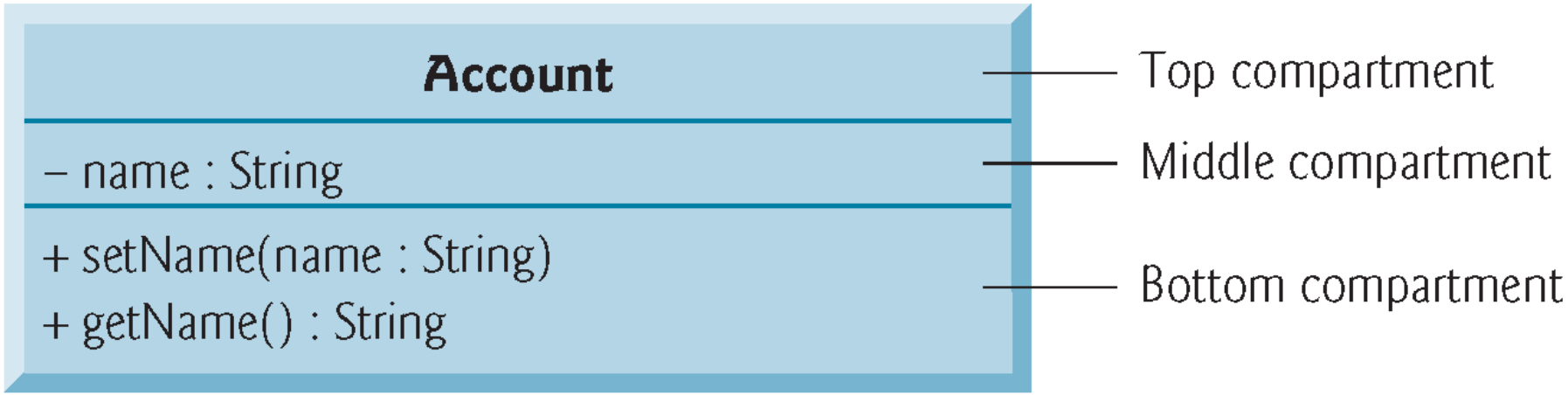
3.2.4 Account UML Class Diagram (Cont.)

Return Types

- The UML indicates an operation's return type by placing a colon and the return type after the parentheses following the operation name.
- UML class diagrams do not specify return types for operations that do not return values.

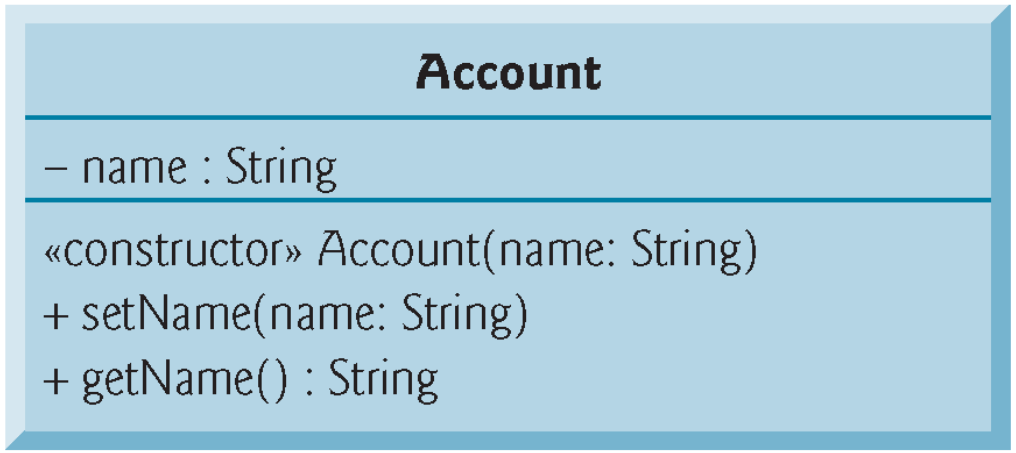
Parameters

- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses after the operation name



Adding the Constructor to Class Account's UML Class Diagram

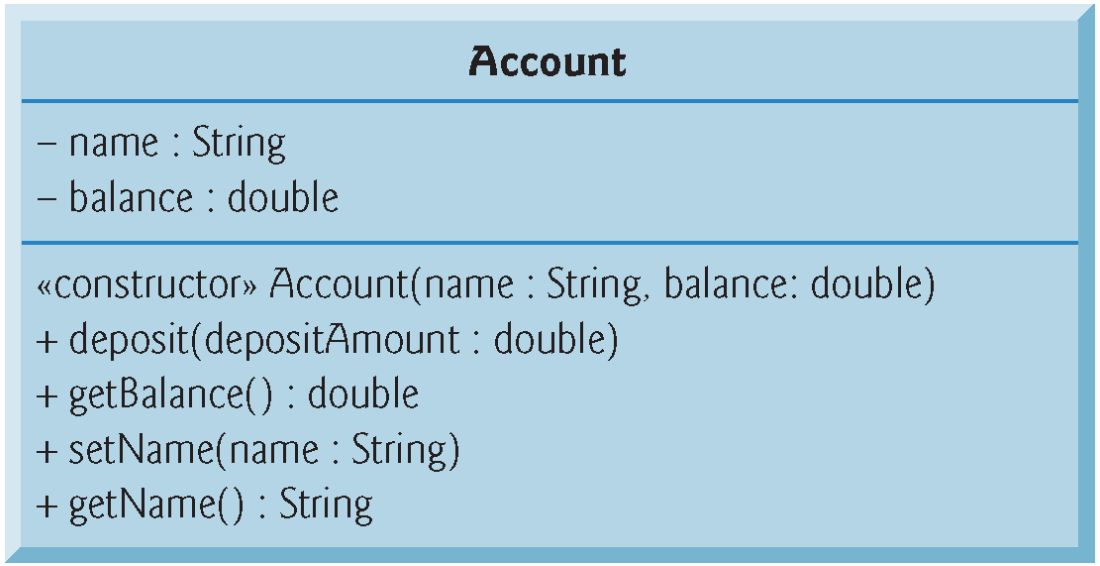
- The UML models **constructors** in the third compartment of a class diagram.
- To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name.




```java
public class Account {
    private String name; // Stores the account holder's name

    public Account(String name) {
        this.name = name;
    }
    // Sets the account holder's name
    public void setName(String name) {
        this.name = name; // Assigns the given name to the instance variable
    }
    // Retrieves the account holder's name
    public String getName() {
        return name; // Returns the stored name
    }
}

public class AccountTest {
    public static void main(String[] args) {
        Account account1 = new Account("Jane Green");
        Account account1 = new Account("John Blue");

        System.out.printf("account1 name is: %s%n ", account1.getName());
        System.out.printf("account2 name is: %s%n ", account2.getName());
    }
}
```

Example

```
// Fig. 3.8: Account.java
public class Account {
   private String name;
   private double balance;
   // Account constructor
   public Account(String name, double balance) {
      this.name = name;

      if (balance > 0.0) {
         this.balance = balance;
      }
   }

   // method that deposits (adds) only a valid amount to the balance
   public void deposit(double depositAmount) {
      if (depositAmount > 0.0) {
         balance = balance + depositAmount;
      }
   }

   // method returns the account balance
   public double getBalance() {

      return balance;
   }

   // method that sets the name
   public void setName(String name) {
      this.name = name;
   }

   // method that returns the name
   public String getName() {
      return name;
   }
}
```



UML Diagram in IntelliJ IDEA

Many IDEs (like IntelliJ IDEA, Eclipse) support UML diagrams, enabling automatic generation and synchronization with code, which saves time.

```
public class PersonUML {
    String name;
    public PersonUML(String name) {
        this.name = name;
    }
    public void introduce() {
        System.out.println("Name is " + name);
    }
}
```

```
public class TestUniversity {
    public static void main(String[] args) {
        StudentUML s = new StudentUML("Alice", 3.8);
        s.introduce();
        s.checkScholarshipEligibility();
    }
}
```

```
public class StudentUML extends PersonUML {
    double gpa;

    public StudentUML(String name, double gpa) {
        super(name);
        this.gpa = gpa;
    }

    public void checkScholarshipEligibility() {
        if (gpa >= 3.5) {
            System.out.println(name + " is eligible for a scholarship.");
        } else {
            System.out.println(name + " is not eligible for a
scholarship.");
        }
    }
}
```

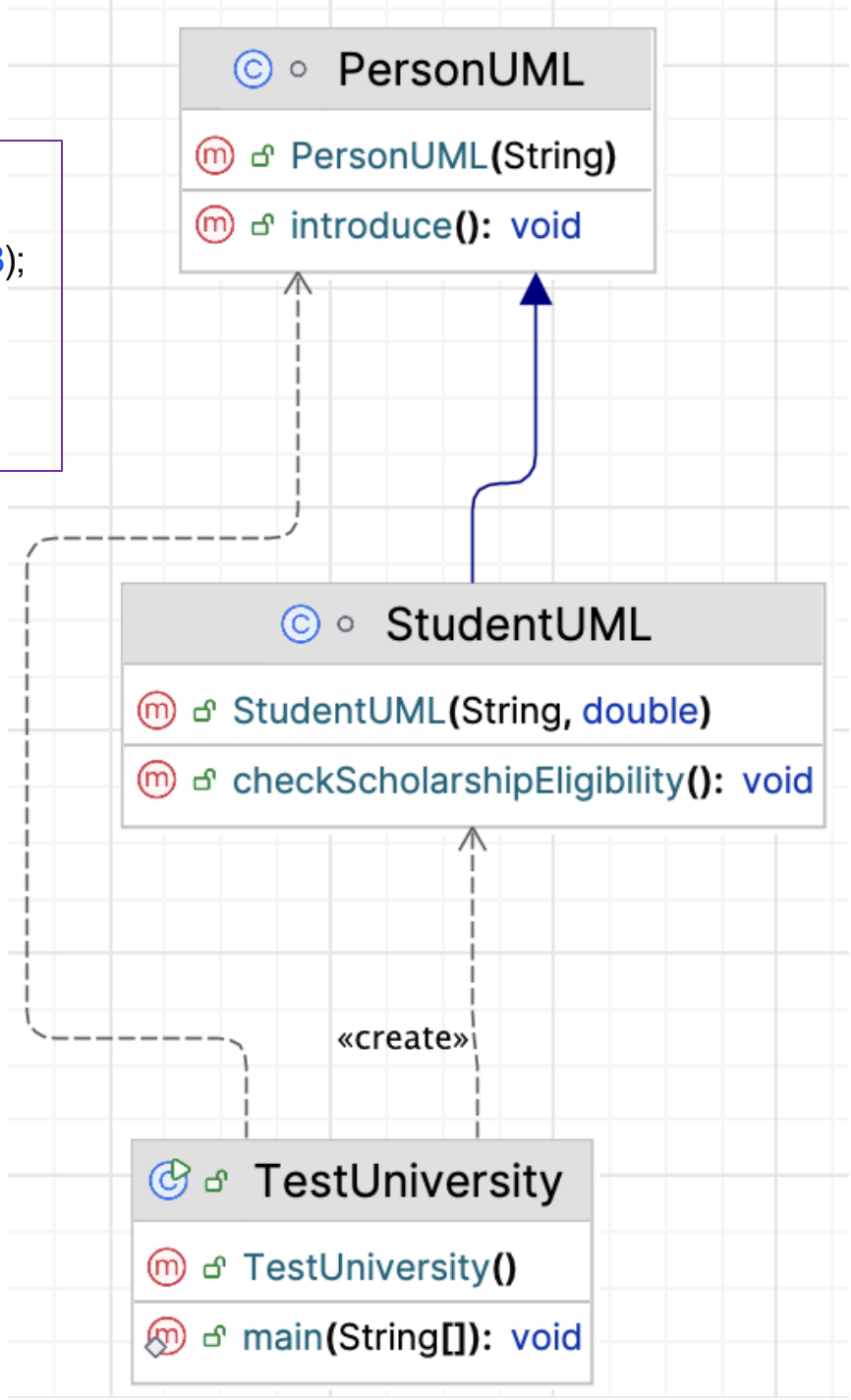
IntelliJ IDEA UML Class Diagrams:
https://www.jetbrains.com/help/idea/2024.2/class-diagram.html?Class_diagram&utm_source=product&utm_medium=link&utm_campaign=IU&utm_content=2024.2&keymap=macOS#manage_class_diagram

UML Diagram in IntelliJ IDEA

```java
public class PersonUML {
    String name;
    public PersonUML(String name) {
        this.name = name;
    }
    public void introduce() {
        System.out.println("Name is " + name);
    }
}
```

```java
public class TestUniversity {
    public static void main(String[] args) {
        StudentUML s = new StudentUML("Alice", 3.8);
        s.introduce();
        s.checkScholarshipEligibility();
    }
}
```

```java
public class StudentUML extends PersonUML {
    double gpa;

    public StudentUML(String name, double gpa) {
        super(name);
        this.gpa = gpa;
    }

    public void checkScholarshipEligibility() {
        if (gpa >= 3.5) {
            System.out.println(name + " is eligible for a scholarship.");
        } else {
            System.out.println(name + " is not eligible for a scholarship.");
        }
    }
}
```



4.4 Control Structures

Sequence Structure in Java

- The activity diagram in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order.
- Models the workflow (also called the activity) of a portion of a software system.
- May include a portion of an algorithm, like the sequence structure in the figure.
- Composed of symbols
  - action-state symbols (rectangles with their left and right sides replaced with outward arcs)
  - diamonds
  - small circles

- Symbols connected by transition arrows, which represent the flow of the activity—the order in which the actions should occur.
- Help you develop and represent algorithms.
- Clearly show how control structures operate.

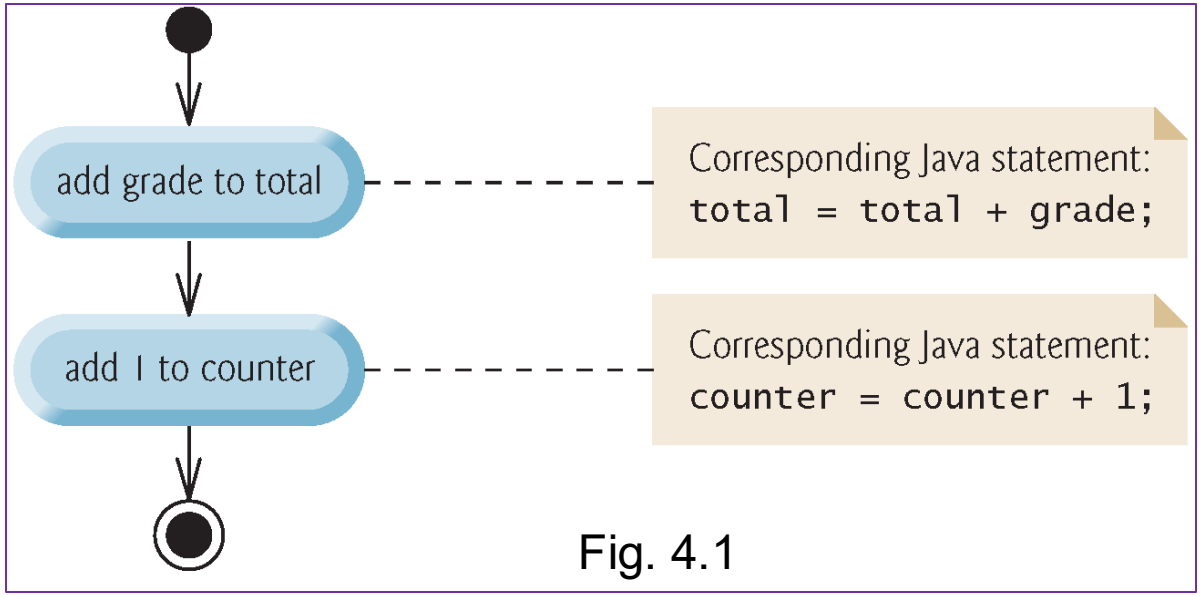
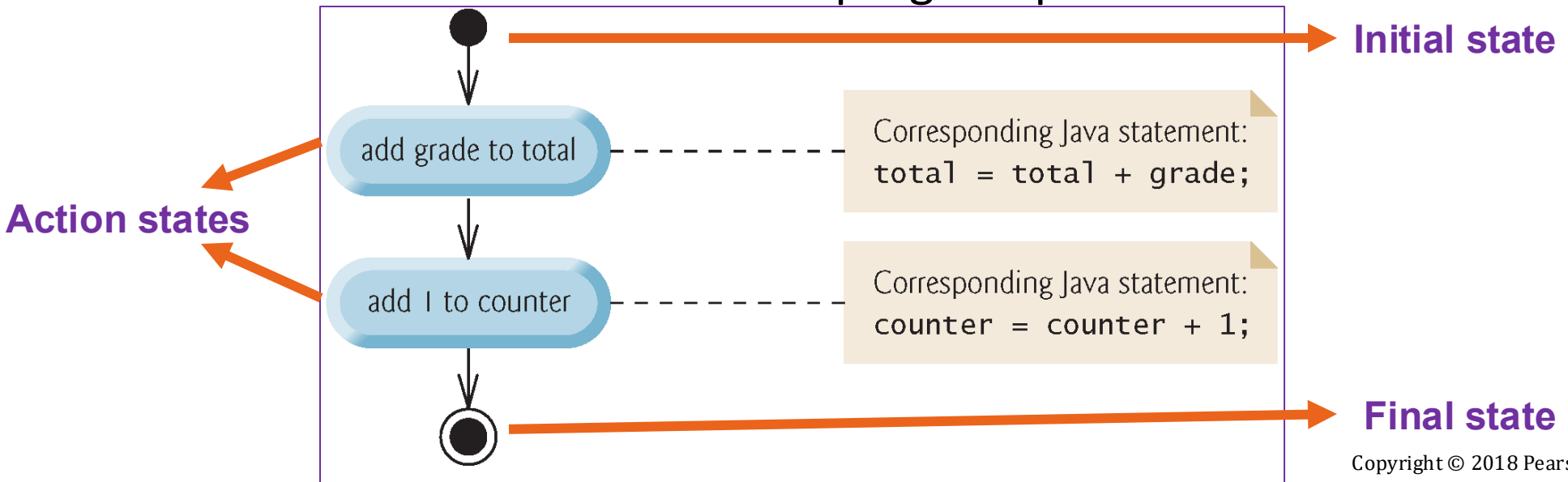


Fig. 4.1

4.4 Control Structures (Cont.)

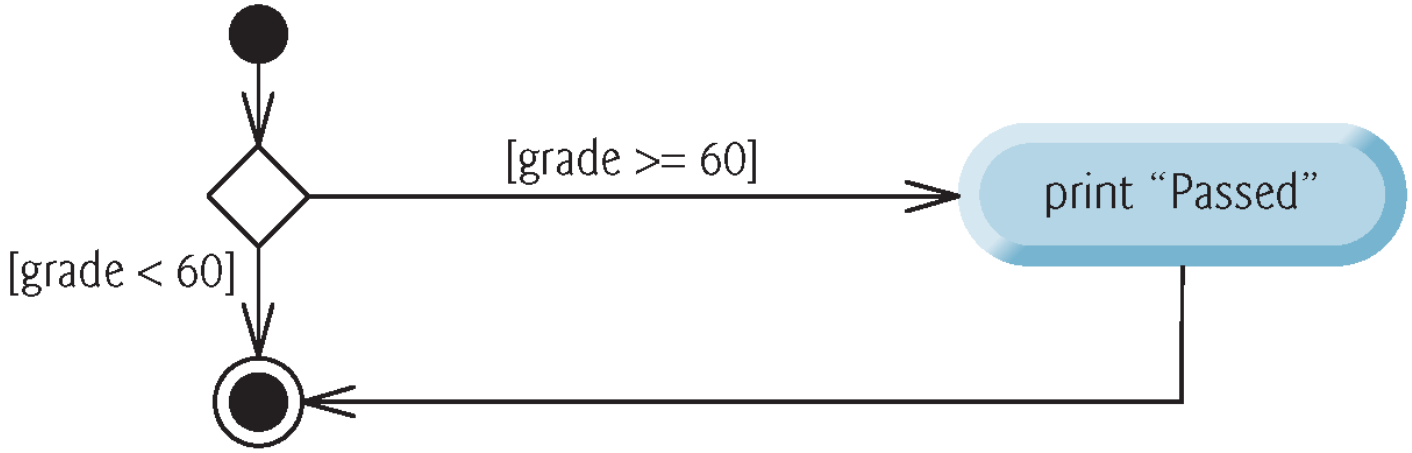
- Sequence-structure activity diagram in Fig. 4.1.
- Two action states that represent actions to perform.
- Each contains an action expression that specifies a particular action to perform.
- Arrows represent transitions (order in which the actions represented by the action states occur).
- Solid circle at the top represents the initial state—the beginning of the workflow before the program performs the modeled actions.
- Solid circle surrounded by a hollow circle at the bottom represents the final state—the end of the workflow after the program performs its actions.



UML Activity Diagram for an if Statement

```
if (studentGrade >= 60) {
    System.out.println("Passed");
}
```

- Diamond, or decision symbol, indicates that a decision is to be made.
- Workflow continues along a path determined by the symbol's guard conditions, which can be true or false.
- Each transition arrow emerging from a decision symbol has a guard condition (in square brackets next to the arrow).
- If a guard condition is true, the workflow enters the action state to which the transition arrow points.



UML Activity Diagram for an if...else Statement

```
    if (grade >= 60) {
       System.out.println("Passed");
    }
    else {
       System.out.println("Failed");
    }
```

- The symbols in the UML activity diagram (besides the initial state, transition arrows and final state) represent action states and decisions.

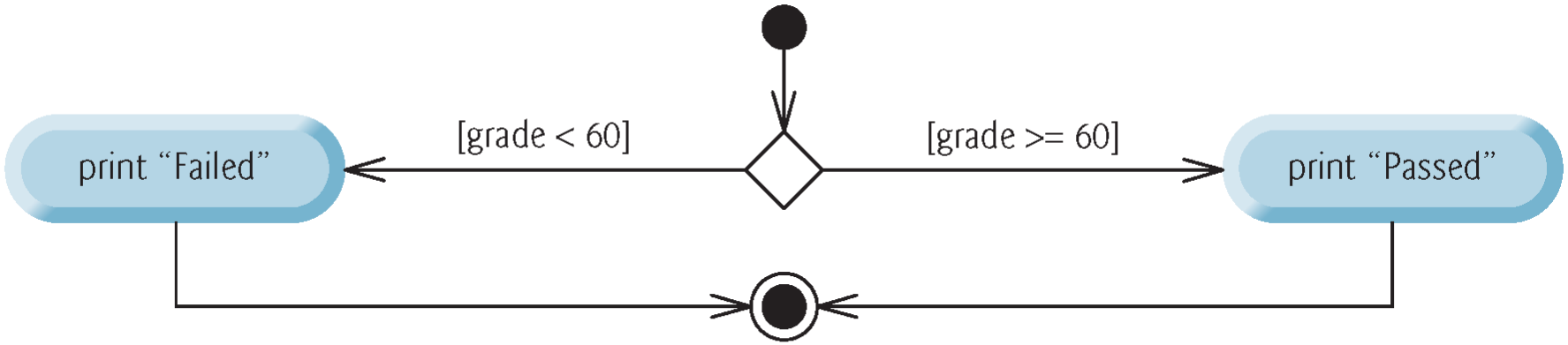
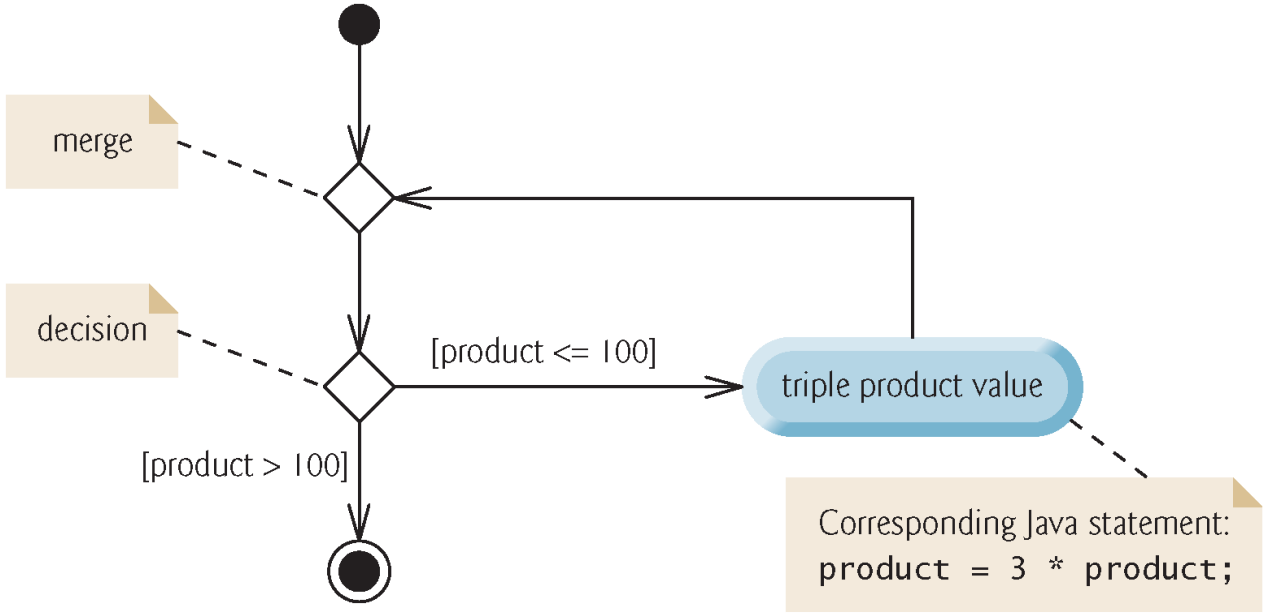


Fig. 4.3 | `if...else` double-selection statement UML activity diagram.

UML Activity Diagram for a while Statement

```
Int product = 3 ;
while (product <= 100){
     product = 3 * product;}
```

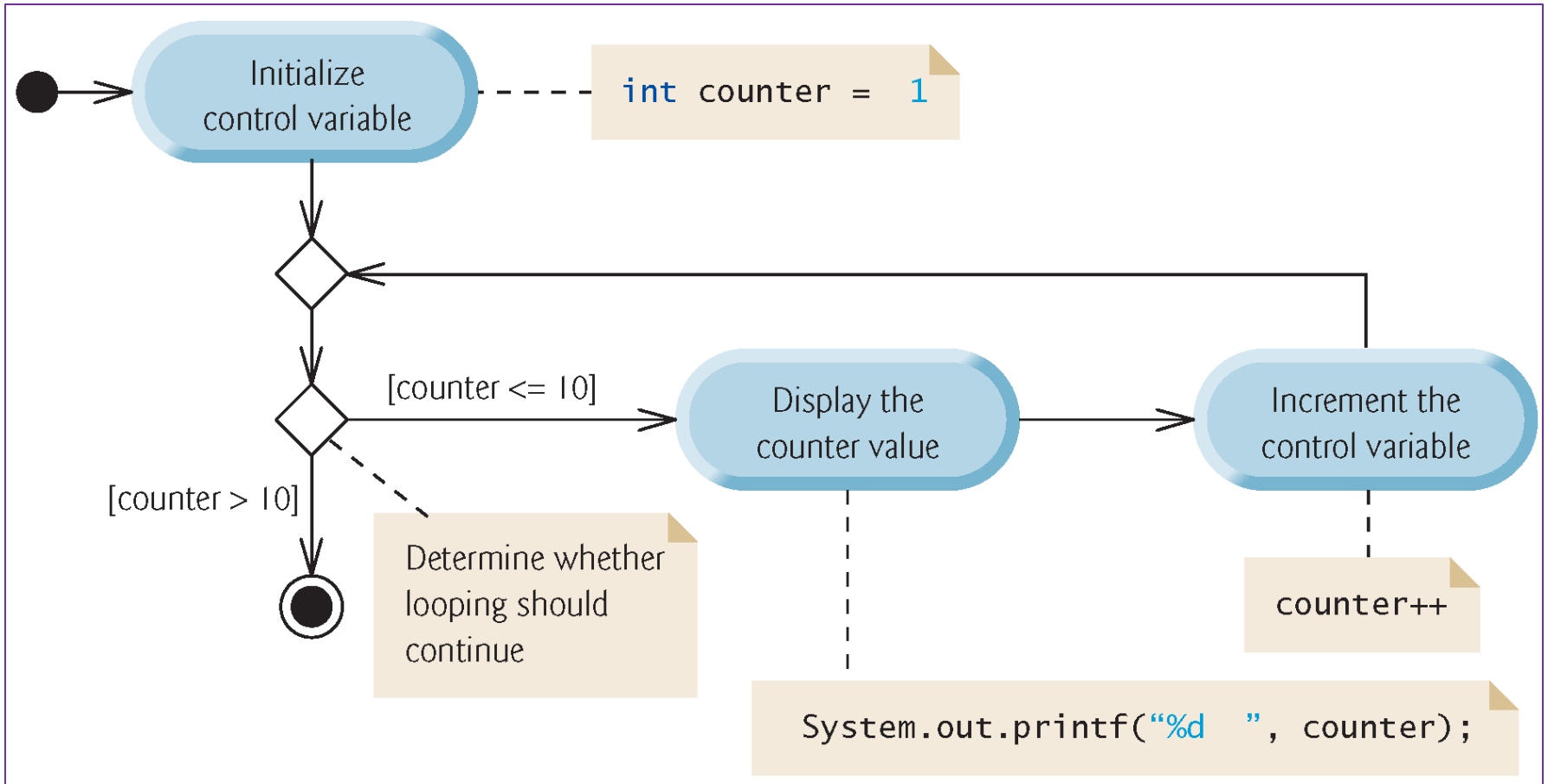
- The UML represents both the merge symbol and the decision symbol as diamonds.
- The merge symbol joins two flows of activity into one.
- The decision and merge symbols can be distinguished by the number of “incoming” and “outgoing” transition arrows.
- A decision symbol has one transition arrow pointing to the diamond and two or more pointing out from it to indicate possible transitions from that point. Each transition arrow pointing out of a decision symbol has a guard condition next to it.
- A merge symbol has two or more transition arrows pointing to the diamond and only one pointing from the diamond, to indicate multiple activity flows merging to continue the activity. None of the transition arrows associated with a merge symbol has a guard condition.



UML Activity Diagram for the `for` Statement

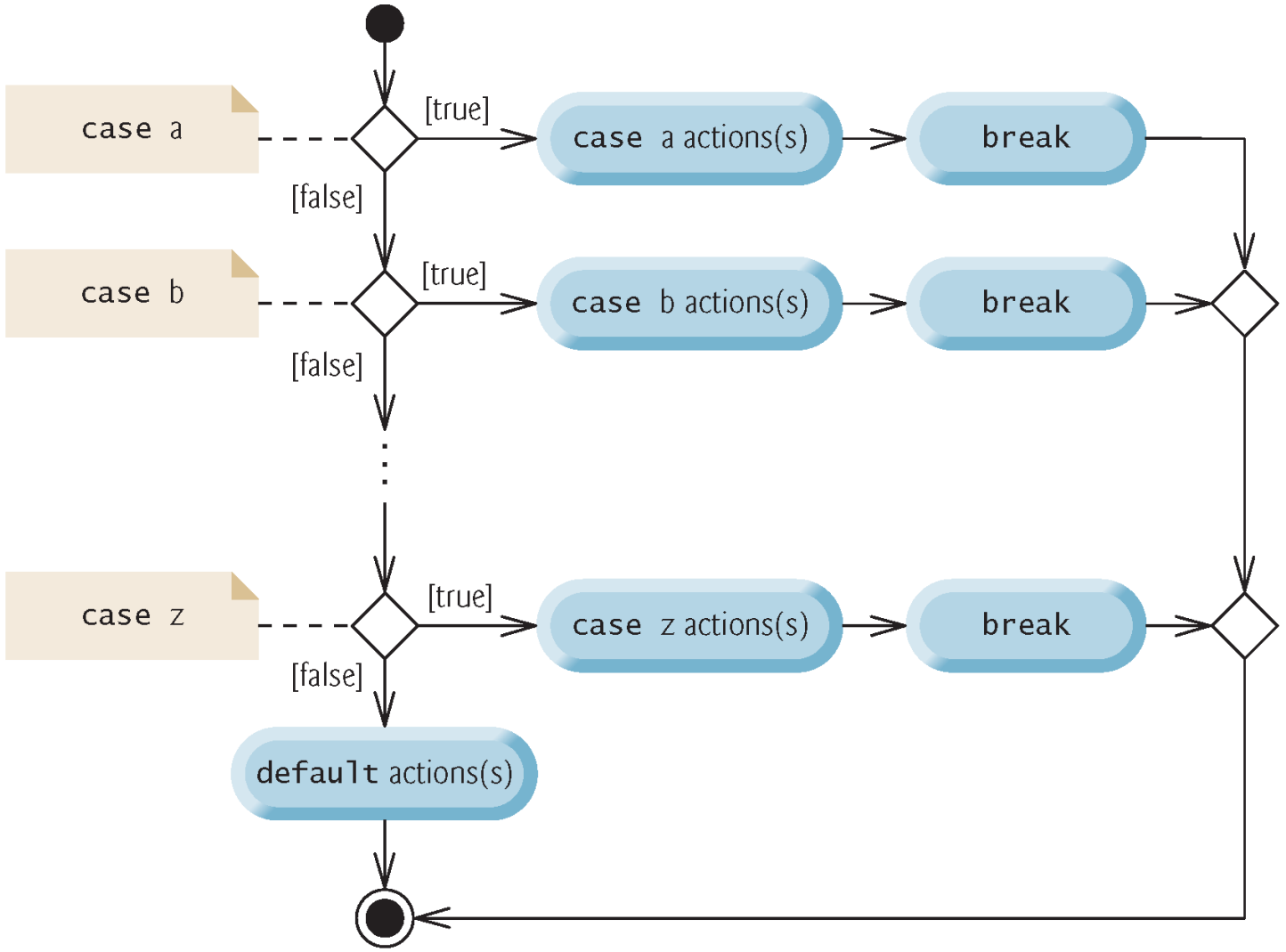
```
public class ForCounter {
        public static void main(String[] args) {
           for (int counter = 1; counter <= 10; counter++) {
              System.out.printf("%d ", counter);
           }
           System.out.println();
    }
}
```

Output: 1 2 3 4 5 6 7 8 9 10



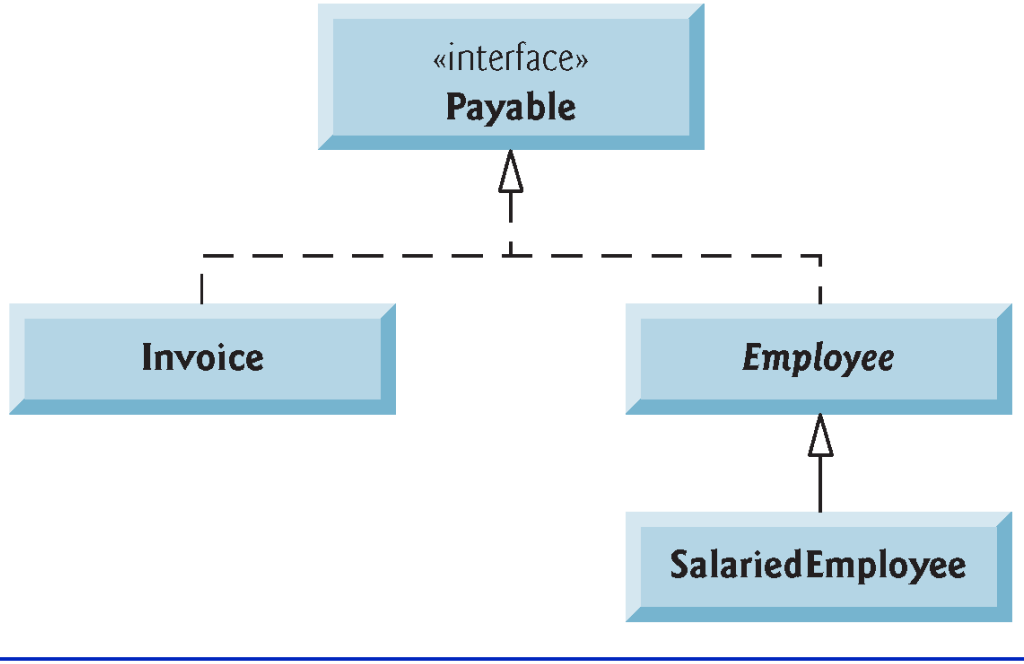
UML activity diagram for the general switch statement

- Most `switch` statements use a `break` in each `case` to terminate the `switch` statement after processing the `case`.
- The `break` statement is not required for the `switch`'s last `case` (or the optional `default` case, when it appears last), because execution continues with the next statement after the `switch`.



10.9.1 Developing a Payable Hierarchy (Cont.)

- Fig. shows the accounts payable hierarchy.
- The UML distinguishes an interface from other classes by placing «interface» above the interface name.



Payable hierarchy UML class diagram.

- The UML expresses the relationship between a class and an interface through a realization.
  - A class is said to "realize," or implement, the methods of an interface.
  - A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface.
- A subclass inherits its superclass's realization relationships.

Introduction to Java Graphical User Interface (GUI)

- **GUI Architecture / GUI Programming Model**
  - Allows users to interact with software visually
  - **Components**
    - Visible **UI elements** (e.g., button, label)
  - **Events**
    - Notifications when a user interacts (invisible)
    - An **object that signals a user action** (e.g., click, key press)
  - **Listeners**
    - Code that reacts to those events
- Java GUI Toolkits
  - **AWT** - Abstract Window Toolkit, older, OS-dependent
  - **Swing** - Lightweight, more features than AWT, platform-independent
  - **JavaFX** - Modern toolkit for rich UIs
- **AWT, Swing and JavaFX do not use the same event and listener classes**, but Swing is largely built on top of AWT's event model, while JavaFX uses its own event system.

AWT (Abstract Window Toolkit)

- First GUI toolkit in Java
- For old/legacy apps or embedded systems
  - An embedded device (e.g., a network router or machine controller) needs a simple interface to set basic parameters — like a name, IP address, and a “Save” button.
    - Printer configuration panel
    - Network settings tool on embedded devices
    - File chooser utility in early Java apps
    - Simple calculator on Java ME/CE devices
- Why Do We Use AWT in Embedded/Legacy Systems?
  - It doesn't need modern UI elements or styling
  - It is suitable for devices with limited memory or CPU
  - It has simple OS-level components
    - These are **basic graphical user interface (GUI)** elements provided directly by the **operating system**, not fancy or custom-designed like in modern applications.
  - It is still part of core Java. No need for extra libraries.

AWT Components

Component	Class Name	Description	Example Usage
Window	Frame	Main top-level window	Frame frame = new Frame("My App");
Panel	Panel	A generic container for components	Panel panel = new Panel();
Label	Label	Displays a short text label	Label label = new Label("Username");
Button	Button	Clickable button	Button btn = new Button("Click Me");
Text Field	TextField	Single-line text input field	TextField tf = new TextField(20);
Text Area	TextArea	Multi-line text input area	TextArea ta = new TextArea(5, 20);
Checkbox	Checkbox	Toggle button (on/off)	Checkbox cb = new Checkbox("Accept");
Radio Button	CheckboxGroup + Checkbox	Used to create radio button groups	CheckboxGroup group = new CheckboxGroup();

Component Package**: java.awt**

AWT Components

Component	Class Name	Description	Example Usage
Choice (Dropdown)	Choice	Dropdown menu for single selection	Choice choice = new Choice();
List	List	List of items (can be single or multiple selection)	List list = new List();
Scrollbar	Scrollbar	Horizontal or vertical scroll bar	Scrollbar sb = new Scrollbar();
Canvas	Canvas	Area for custom drawing (used in games/graphics)	Canvas canvas = new Canvas();
MenuBar	MenuBar	Top-level menu bar	MenuBar mb = new MenuBar();
Menu	Menu, MenuItem	Dropdown menu and menu item	Menu file = new Menu("File");
Dialog	Dialog	Popup dialog window	Dialog d = new Dialog(frame, "Message");

Component Package**: java.awt**

AWT - Event and Listener

Event Type	Event Class	Listener Interface	Triggered When...
Action Event	ActionEvent	ActionListener	A button is clicked, a menu item is selected, etc.
Item Event	ItemEvent	ItemListener	An item (like checkbox or choice) is selected/deselected
Adjustment Event	AdjustmentEvent	AdjustmentListener	A scrollbar is moved
Text Event	TextEvent	TextListener	Text is changed in a text component (TextField, TextArea)
Key Event	KeyEvent	KeyListener	A key is pressed, released, or typed
Mouse Event	MouseEvent	MouseListener	Mouse is clicked, pressed, released, entered, exited
Mouse Event	MouseEvent	MouseMotionListener	Mouse is moved or dragged
Window Event	WindowEvent	WindowListener	A window is opened, closed, iconified, activated, etc.
Focus Event	FocusEvent	FocusListener	A component gains or loses focus
Component Event	ComponentEvent	ComponentListener	Component is resized, moved, shown, or hidden
Container Event	ContainerEvent	ContainerListener	A component is added to or removed from a container
Paint Event	PaintEvent	(Handled automatically)	Component needs to be redrawn
Input Method Event	InputMethodEvent	InputMethodListener	Text input using input methods (e.g., for languages like Japanese)

Event and Listener Package: **java.awt.event**

Example: AWT

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

public class DeviceSettingsAWT {
    public static void main(String[] args) {
        Frame frame = new Frame("Device Settings");

        // Create labels and fields
        Label nameLabel = new Label("Device Name:");
        TextField nameField = new TextField(20);

        Label ipLabel = new Label("IP Address:");
        TextField ipField = new TextField(20);

        Button saveButton = new Button("Save");

        // Set layout and add components
        frame.setLayout(new FlowLayout());
        //Adds the device name label and text field to the frame.
        frame.add(nameLabel);
        frame.add(nameField);

        //Adds the IP address label and text field to the frame.
        frame.add(ipLabel);
        frame.add(ipField);

        frame.add(saveButton); //Adds the "Save" button to the frame.
```

Import all classes for GUI components like Frame, Label, Button, etc.

Import all classes for event handling, like ActionListener, WindowAdapter.

Creates a new AWT window (Frame) with the title "Device Settings".

Creates a label component displaying "Device Name:".

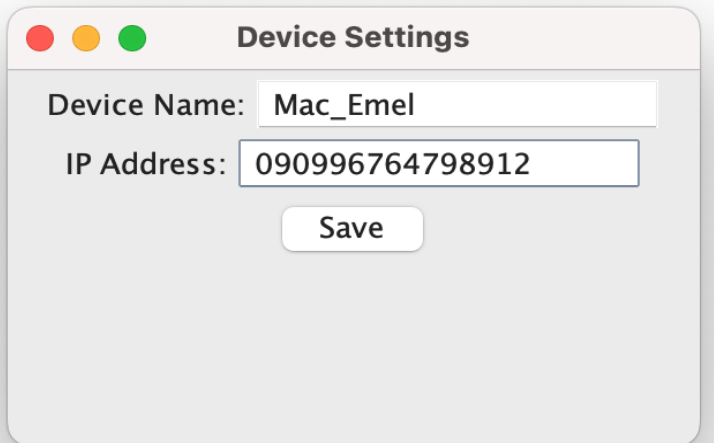
Creates a text input field (TextField) with a width of 20 columns for user input.

Creates another label with "IP Address:".

Creates another text input field for entering the IP address, also 20 columns wide.

Creates a clickable button labeled "Save".

Sets the layout manager of the frame to FlowLayout. FlowLayout arranges added components left to right in the order they are added, wrapping to new lines as needed.



Example: AWT

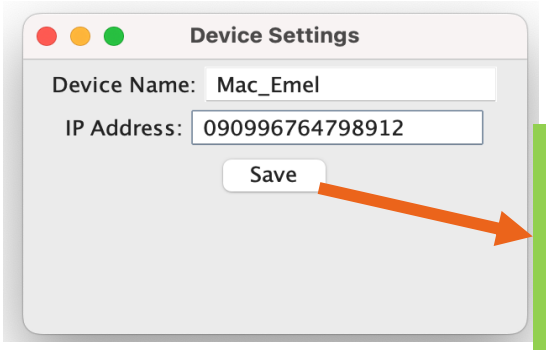
```
    // Add action listener
    saveButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            String name = nameField.getText();
            String ip = ipField.getText();
            System.out.println("Saved:\nName: " + name + "\nIP: " + ip);
        }
    });

    // Configure frame
    frame.setSize(300, 200);
    frame.setVisible(true);

    // Add close window functionality
    frame.addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            frame.dispose();
        }
    });
}
}
```

- Adds an **action listener** to the save button.
- When the button is clicked, the actionPerformed method is called.
- Retrieves text from the name and IP fields.
- Prints the values to the console.

Sets the window size to 300 pixels wide by 200 pixels tall.

Makes the window visible on the screen.

- Adds a window listener to the frame using a subclass of WindowAdapter.
  - WindowAdapter is a convenience class that implements the WindowListener interface with empty method bodies, so you don't have to override all 7 methods if you only care about one.
- Overrides windowClosing method to call frame.dispose().
- When the user clicks the window's close button, this code releases the window resources and closes the window properly.



Output:
Saved:
Name: Mac_Emel
IP: 090996764798912

Swing

- For desktop apps with moderate complexity
  - Inventory management system
  - Student record management
  - Budget tracker
  - Address book

- Improved over AWT
- Tutorials: https://docs.oracle.com/javase/tutorial/uiswing/

Swing Components

Component	Class Name	Description	Example Usage
Window	JFrame	Main application window	JFrame frame = new JFrame("My App");
Panel	JPanel	Container to group components	JPanel panel = new JPanel();
Label	JLabel	Displays a short text or image	JLabel label = new JLabel("Username");
Button	JButton	Clickable button	JButton btn = new JButton("Login");
Text Field	JTextField	Single-line input field	JTextField tf = new JTextField(10);
Password Field	JPasswordField	Input field for passwords (masked text)	JPasswordField pf = new JPasswordField();
Text Area	JTextArea	Multi-line text input area	JTextArea ta = new JTextArea(5, 20);
Check Box	JCheckBox	Toggleable box for boolean choice	JCheckBox cb = new JCheckBox("Accept");
Radio Button	JRadioButton	Select one from a group of options	JRadioButton rb = new JRadioButton("Yes");

Component Package: **javax.swing**

Swing Components

Component	Class Name	Description	Example Usage
Button Group	ButtonGroup	Groups radio buttons to allow single selection	ButtonGroup group = new ButtonGroup();
Combo Box	JComboBox	Drop-down menu	JComboBox cb = new JComboBox(items);
List	JList	Displays a list of items	JList list = new JList(data);
Table	JTable	Displays data in rows and columns	JTable table = new JTable(data, columns);
Menu Bar	JMenuBar	Top bar with menus	JMenuBar menuBar = new JMenuBar();
Menu	JMenu, JMenuItem	Dropdown menu and its items	JMenu file = new JMenu("File");
Scroll Pane	JScrollPane	Adds scrollbars to components	new JScrollPane(textArea);
Tabbed Pane	JTabbedPane	Allows tabbed sections	JTabbedPane tabs = new JTabbedPane();
Slider	JSlider	Slide bar for numeric input	JSlider slider = new JSlider();
Progress Bar	JProgressBar	Shows progress of tasks	JProgressBar pb = new JProgressBar();
Dialog	JOptionPane	Popup dialog for messages, input, confirm	JOptionPane.showMessageDialog(null, "Hi");

Component Package: **javax.swing**

Swing Event and Listener

Event Type	Event Class	Listener Interface	Package	Description
Action Event	ActionEvent	ActionListener	java.awt.event	Button clicks, menu items, etc.
Mouse Event	MouseEvent	MouseListener	java.awt.event	Mouse click, press, release, enter, exit
Mouse Motion Event	MouseEvent	MouseMotionListener	java.awt.event	Mouse move and drag
Key Event	KeyEvent	KeyListener	java.awt.event	Keyboard typing and key presses
Focus Event	FocusEvent	FocusListener	java.awt.event	Gaining or losing keyboard focus
Window Event	WindowEvent	WindowListener	java.awt.event	Opening, closing, iconifying, etc.
Component Event	ComponentEvent	ComponentListener	java.awt.event	Component moved, resized, shown, hidden
Item Event	ItemEvent	ItemListener	java.awt.event	Combo boxes, checkboxes, item selection
Adjustment Event	AdjustmentEvent	AdjustmentListener	java.awt.event	Scrollbar adjustments
Change Event	ChangeEvent	ChangeListener	javax.swing.event	Sliders, spinners, tabbed panes, etc.
List Selection Event	ListSelectionEvent	ListSelectionListener	javax.swing.event	Item selection in JList
Document Event	DocumentEvent	DocumentListener	javax.swing.event	Changes to text in text components
Tree Selection Event	TreeSelectionEvent	TreeSelectionListener	javax.swing.event	Node selection in JTree
Table Model Event	TableModelEvent	TableModelListener	javax.swing.event	Changes in JTable data

Event and Listener Package: **Mostly java.awt.event, plus some in javax.swing.event**

Example: Swing Components

```
import javax.swing.*;

public class LoginForm {
    public static void main(String[] args) {
        // Create the main frame
        JFrame frame = new JFrame("Login");
        // Create a panel to hold the components
        JPanel panel = new JPanel();
        // Add components to the panel
        panel.add(new JLabel("Username:"));
        panel.add(new JTextField(10));
        panel.add(new JLabel("Password:"));
        panel.add(new JPasswordField(10));

        panel.add(new JButton("Login"));
        // Add panel to frame
        frame.add(panel);

        // Set default close operation
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Pack the frame to fit components
        frame.pack();

        // Set frame visibility
        frame.setVisible(true);
    }
}
```

imports all Swing classes (like JFrame, JPanel, JLabel, JTextField, etc.) from the javax.swing package.

- Creates a new window (JFrame) titled **"Login"**.
- This is the main container for your GUI.

Creates a **panel** to hold and organize the GUI components. A JPanel is like a layout board you attach elements to.

Adds two **labels** with the text "Username:" and "Password:" to the panel.

Adds a **text field** where the user can enter their username. The number 10 is the column size (approximate width of the field).

Adds a **password field** for entering hidden text (e.g., dots instead of characters). The size is 10 columns.

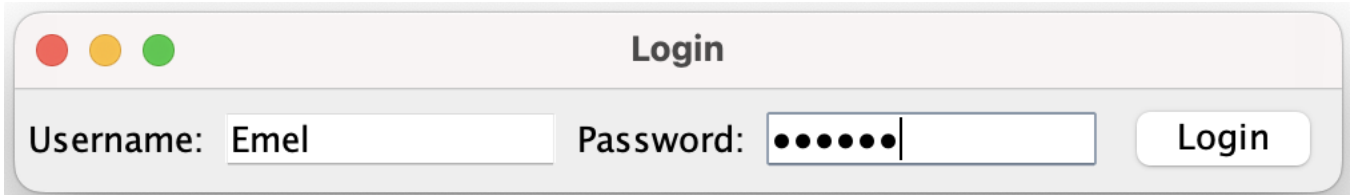
Adds a **button** labeled **"Login"**.

Adds the **panel** (with all the components) to the **frame** (main window).

Ensures the application **closes completely** when the window is closed. Without this, the program may keep running in the background.

Automatically sizes the window to **fit all components neatly**. It avoids manual width/height settings.

Makes the window **visible** to the user. If you skip this line, the window won't show up.



JavaFX

- For modern, stylish UIs, or when CSS/media/animation needed
- JavaFX uses Stage, Scene, Nodes
- Supports 2D/3D graphics, charts, media
- JavaFX components are more modern and **hardware-accelerated**.
- JavaFX is preferred for building **rich desktop applications**.
  - Music player
  - Task planner with calendar
  - Video library organizer
  - Dashboard for IoT sensors
- Turorials: https://openjfx.io/openjfx-docs/#introduction

JavaFX Components

Component	Class Name	Description	Example Usage
Label	Label	Displays non-editable text	new Label("Name:")
Button	Button	Clickable button	new Button("Submit")
Text Field	TextField	Single-line user text input	new TextField()
Password Field	PasswordField	Hides user input (e.g., for passwords)	new PasswordField()
Text Area	TextArea	Multi-line text input	new TextArea()
CheckBox	CheckBox	A checkbox (on/off)	new CheckBox("Accept Terms")
Radio Button	RadioButton	One option in a group of mutually exclusive options	new RadioButton("Male")
ToggleGroup	ToggleGroup	Groups radio buttons so only one can be selected	radio.setToggleGroup(group);
ComboBox	ComboBox	Drop-down menu with selectable options	new ComboBox<String>()
ListView	ListView	Scrollable list of items	new ListView<String>()
TableView	TableView	Displays data in tabular format	new TableView<Person>()
TreeView	TreeView	Displays hierarchical (tree-like) data	new TreeView<String>()
Slider	Slider	Select a value from a range using a sliding knob	new Slider(0, 100, 50)
ProgressBar	ProgressBar	Shows progress of a task (horizontal)	new ProgressBar(0.5)
ProgressIndicator	ProgressIndicator	Circular progress indicator	new ProgressIndicator(0.5)
DatePicker	DatePicker	Input component for selecting dates	new DatePicker()
ColorPicker	ColorPicker	Let users choose a color	new ColorPicker()

Component Package: **javafx.scene.control, javafx.scene.***

JavaFX Components

Component	Class Name	Description	Example Usage
ImageView	ImageView	Displays an image	new ImageView(new Image("photo.jpg"))
WebView	WebView	Displays web content using an embedded browser	new WebView()
MenuBar	MenuBar	Top-level menu container	new MenuBar()
TabPane	TabPane	Container with tabbed navigation	new TabPane()
Accordion	Accordion	Expandable/collapsible content panels	new Accordion()
ToolBar	ToolBar	Horizontal bar with buttons and tools	new ToolBar(button1, button2)
AnchorPane	AnchorPane	Layout where nodes are anchored to the edges	new AnchorPane()
VBox / HBox	VBox, HBox	Vertical / horizontal layout containers	new VBox(node1, node2)
GridPane	GridPane	Grid layout manager (rows and columns)	new GridPane()
BorderPane	BorderPane	Layout with top, bottom, left, right, and center areas	new BorderPane()
StackPane	StackPane	Stacks nodes on top of each other	new StackPane()
Scene	Scene	The container for all content in a window	new Scene(rootPane, 400, 300)
Stage	Stage	Represents the main window or a dialog	primaryStage.setScene(scene);

Component Package: **javafx.scene.control, javafx.scene.***

JavaFX Event and Listener

Event Type	Event Class	Handler Type	Typical Use
Action Event	javafx.event.ActionEvent	EventHandler<ActionEvent>	Button clicks, menu selections, etc.
Mouse Event	javafx.scene.input.MouseEvent	EventHandler<MouseEvent>	Mouse click, drag, move, etc.
Key Event	javafx.scene.input.KeyEvent	EventHandler<KeyEvent>	Keyboard input events
Drag Event	javafx.scene.input.DragEvent	EventHandler<DragEvent>	Drag and drop interactions
Touch Event	javafx.scene.input.TouchEvent	EventHandler<TouchEvent>	Touchscreen interactions
Scroll Event	javafx.scene.input.ScrollEvent	EventHandler<ScrollEvent>	Mouse wheel scroll or touchpad scrolling
Zoom Event	javafx.scene.input.ZoomEvent	EventHandler<ZoomEvent>	Pinch zoom gestures
Swipe Event	javafx.scene.input.SwipeEvent	EventHandler<SwipeEvent>	Swipe gestures
Rotate Event	javafx.scene.input.RotateEvent	EventHandler<RotateEvent>	Two-finger rotation gestures
Window Event	javafx.stage.WindowEvent	EventHandler<WindowEvent>	Window shown, hidden, closing, etc.
Input Method Event	javafx.scene.input.InputMethodEvent	EventHandler<InputMethodEvent>	Text input from input methods (e.g., IME)
Context Menu Event	javafx.scene.input.ContextMenuEvent	EventHandler<ContextMenuEvent>	Right-click or touch-and-hold events

Event and Listener Package**: javafx.event, javafx.event.EventHandler**

Example JavaFX

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import javafx.event.ActionEvent;
import javafx.event.EventHandler;

public class ClickCounterApp extends Application {

    private int count = 0;  // Counter variable

    @Override
    public void start(Stage stage) {
        Label label = new Label("Clicked 0 times");
        Button button = new Button("Click me");

        button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent e) {
                count++;  // Increase counter
                label.setText("Clicked " + count + " times");
            }
        });
```

Imports the **Application** class, the base for all JavaFX apps. Your app extends this class.

Imports the **Scene** class, the container for all UI elements shown inside a window.

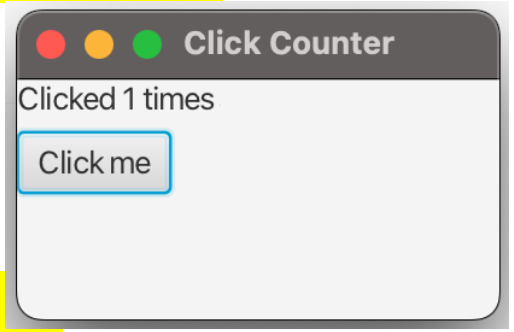
Imports JavaFX **UI controls**: Button and Label.

Imports **VBox layout**, which arranges UI elements vertically.

Imports the **Stage** class representing the main app window.

Imports event classes:

- **ActionEvent**: Represents events like button clicks.
- **EventHandler**: Interface for handling events.



Declares the ClickCounterApp class that **extends Application**, making it a JavaFX app.

The start method is called when the app launches. Stage represents the window.

Creates a **Label** that initially shows the text "Clicked 0 times".

Creates a **Button** labeled "Click me".

Adds an **event listener** for button clicks:

- Uses an anonymous class implementing EventHandler<ActionEvent>.
- The handle method runs on each button click.
- Increments count.
- Updates the label text to show the current count.

Example JavaFX

```
    VBox root = new VBox(10, label, button);
    Scene scene = new Scene(root, 100, 150);

    stage.setScene(scene);
    stage.setTitle("Click Counter");
    stage.show();
  }

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

Creates a **VBox layout** with 10 pixels spacing.
Adds the label and button vertically stacked.

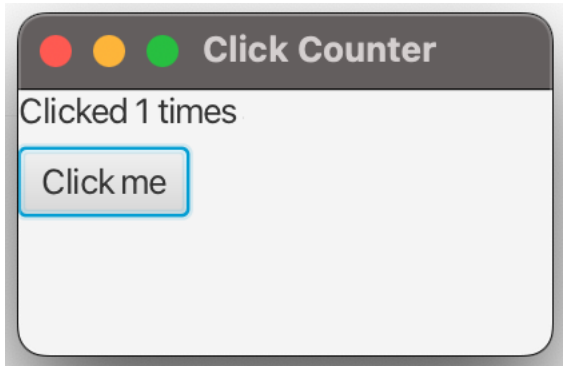
Creates a **Scene** using the VBox root layout.
Sets window size to 100x150 pixels.

Sets the scene on the main application window (stage).

Sets the window title to "Click Counter".

Displays the window.

The main method calls launch(), which starts the JavaFX runtime and eventually calls start().



Questions?