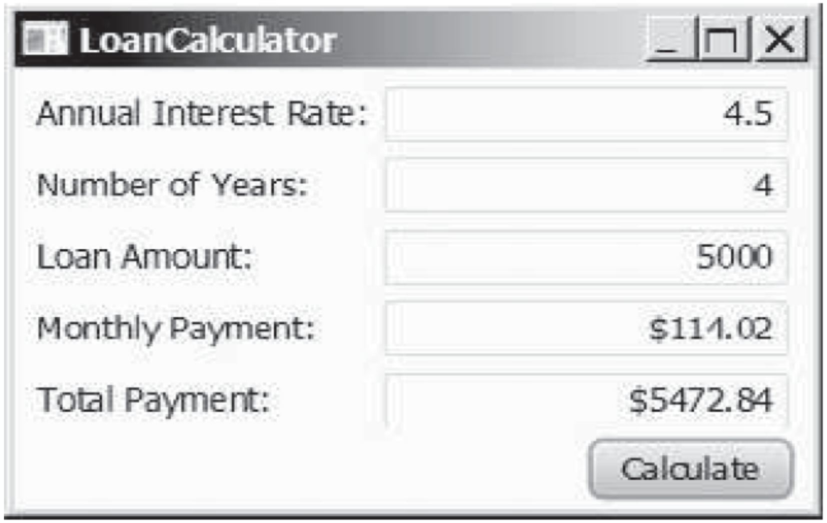
§15.1 Introduction to Event-Driven Software

- In *procedural code,* the program's flow execution is determined by the program's structure (and perhaps its input)
- In *event-driven code*, the *user* is responsible for determining what happens next
- GUIs use event-driven programming to respond to events:
  - Clicking on a button or Selecting a check box or a radio button

- Suppose you want to develop a GUI application to calculate loan payments:
- The user should be able type the pertinent values into the boxes, and then click "calculate"
- How can our program tell when the "Calculate" button has been pressed (clicked on), so we can run the code to *do* the calculation?



Annual Interest Rate: 4.5

Number of Years: 4

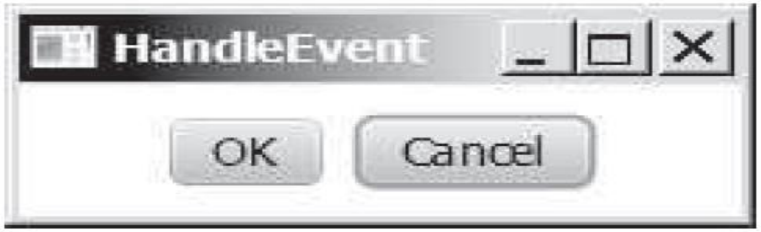
Loan Amount: 5000

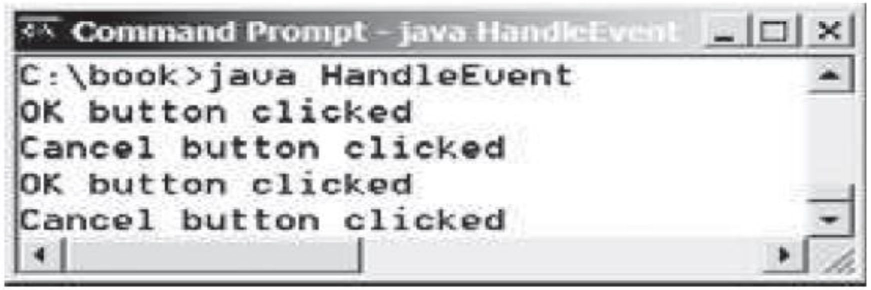
Monthly Payment: $114.02

Total Payment: $5472.84

§15.1 Introduction to Event-Driven Software

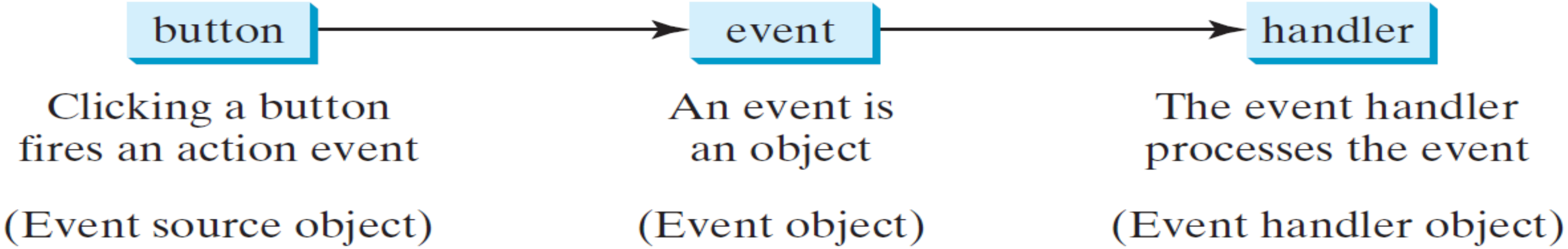
- We'll create a stage with two buttons – "OK" and "Cancel"





```
C:\book>java HandleEvent
OK button clicked
Cancel button clicked
OK button clicked
Cancel button clicked
```

- To respond to a button click, you need to write the code to process the button-clicking action.
- The button is an *event source object* – where the action originates.
- You need to create an object capable of handling the action (click) event on a button.
- This object is called an *event handler*, as shown in Figure 15.3



Clicking a button fires an action event

(Event source object)

An event is an object

(Event object)

The event handler processes the event

(Event handler object)

Event Handler

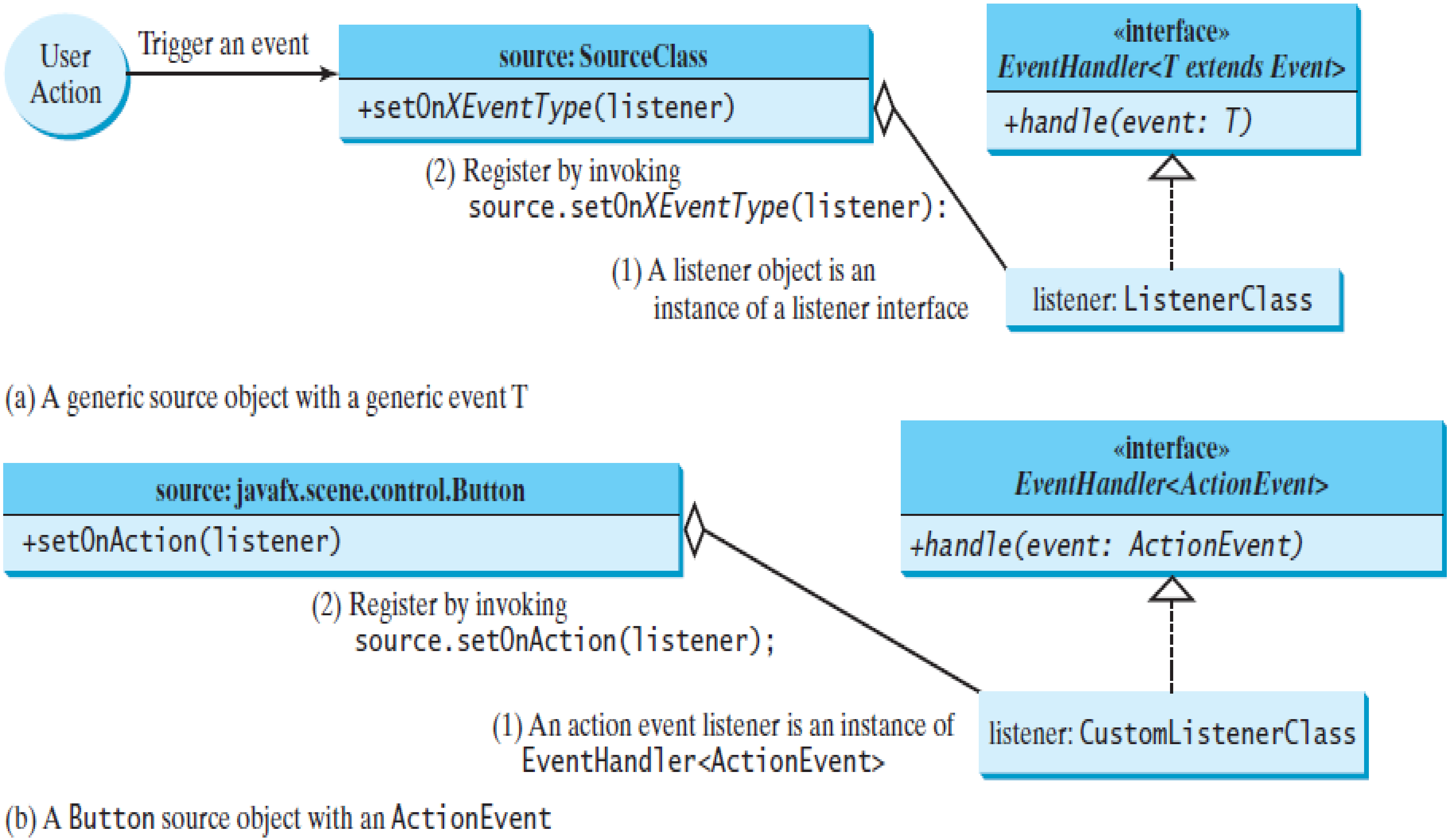
- To be an action event handler:

1. The object must implement the interface EventHandler<T extends Event>, which defines the common behavior for all action handlers
2. The EventHandler object handler must be ***registered*** with the event source object using the source.setOnAction(handler) method

- The EventHandler<ActionEvent> interface contains the method handle(ActionEvent) for processing the event -- your handler class ***must override*** this method to respond to the event

```
// Create the button line 16):
Button btOK = new Button("OK");
// Create handler to receive button’s events (18):
OKHandlerClass handler1 = new OKHandlerClass();
// Register the handler with the button (line 19)
// This tells the button where to send ActionEvents
btOK.setOnAction(handler1);
```

Registering Handlers

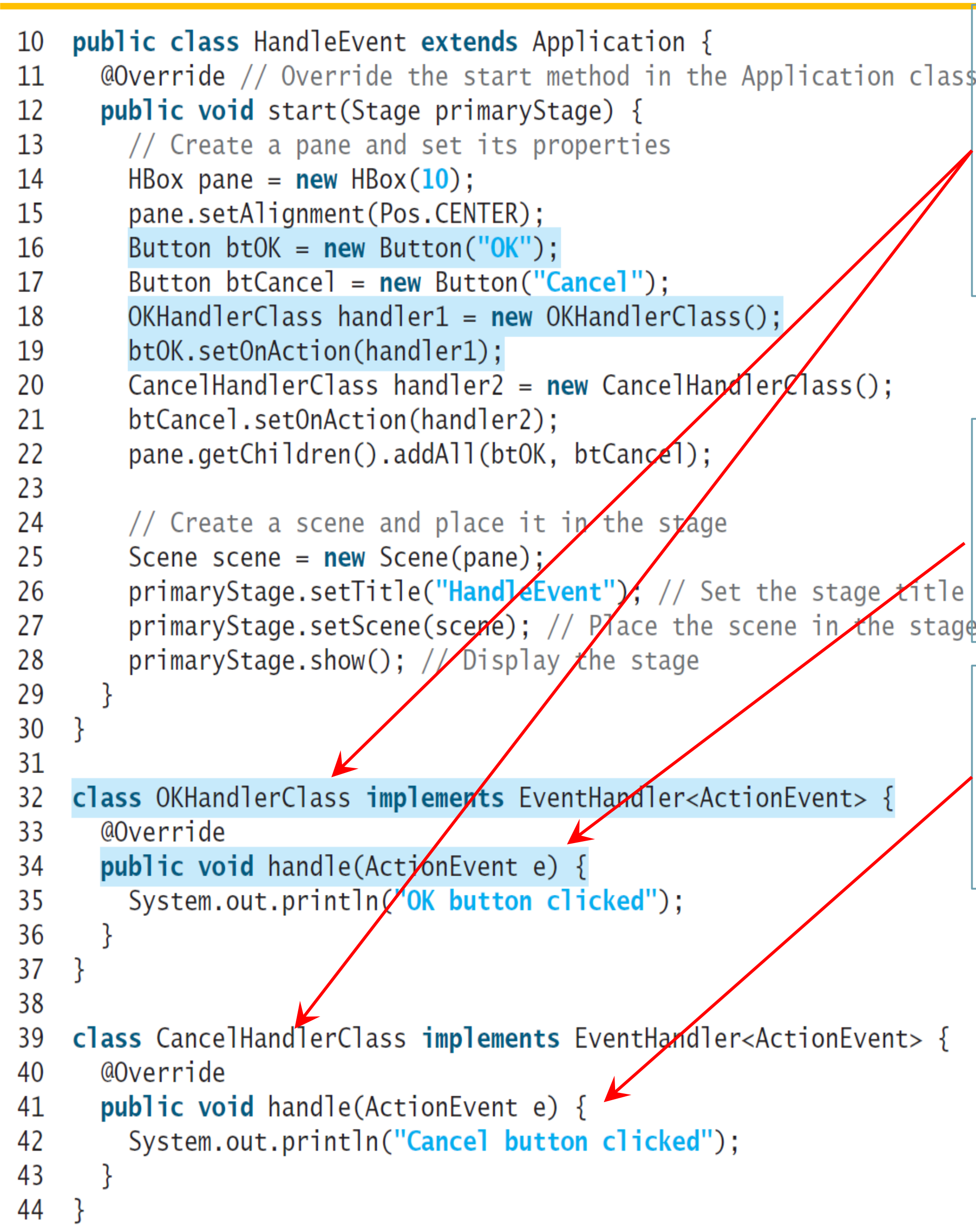

User Action
Trigger an event
source: SourceClass
+setOnXEventType(listener)
«interface»
EventHandler<T extends Event>
+handle(event: T)
(2) Register by invoking
source.setOnXEventType(listener);
(1) A listener object is an
instance of a listener interface
listener: ListenerClass
source: javafx.scene.control.Button
+setOnAction(listener)
«interface»
EventHandler<ActionEvent>
+handle(event: ActionEvent)
(2) Register by invoking
source.setOnAction(listener);
(1) An action event listener is an instance of
EventHandler<ActionEvent>
listener: CustomListenerClass

(a) A generic source object with a generic event T

(b) A Button source object with an ActionEvent

FIGURE 15.5 A listener must be an instance of a listener interface and must be registered with a source object.

LISTING 15.1 HandleEvent



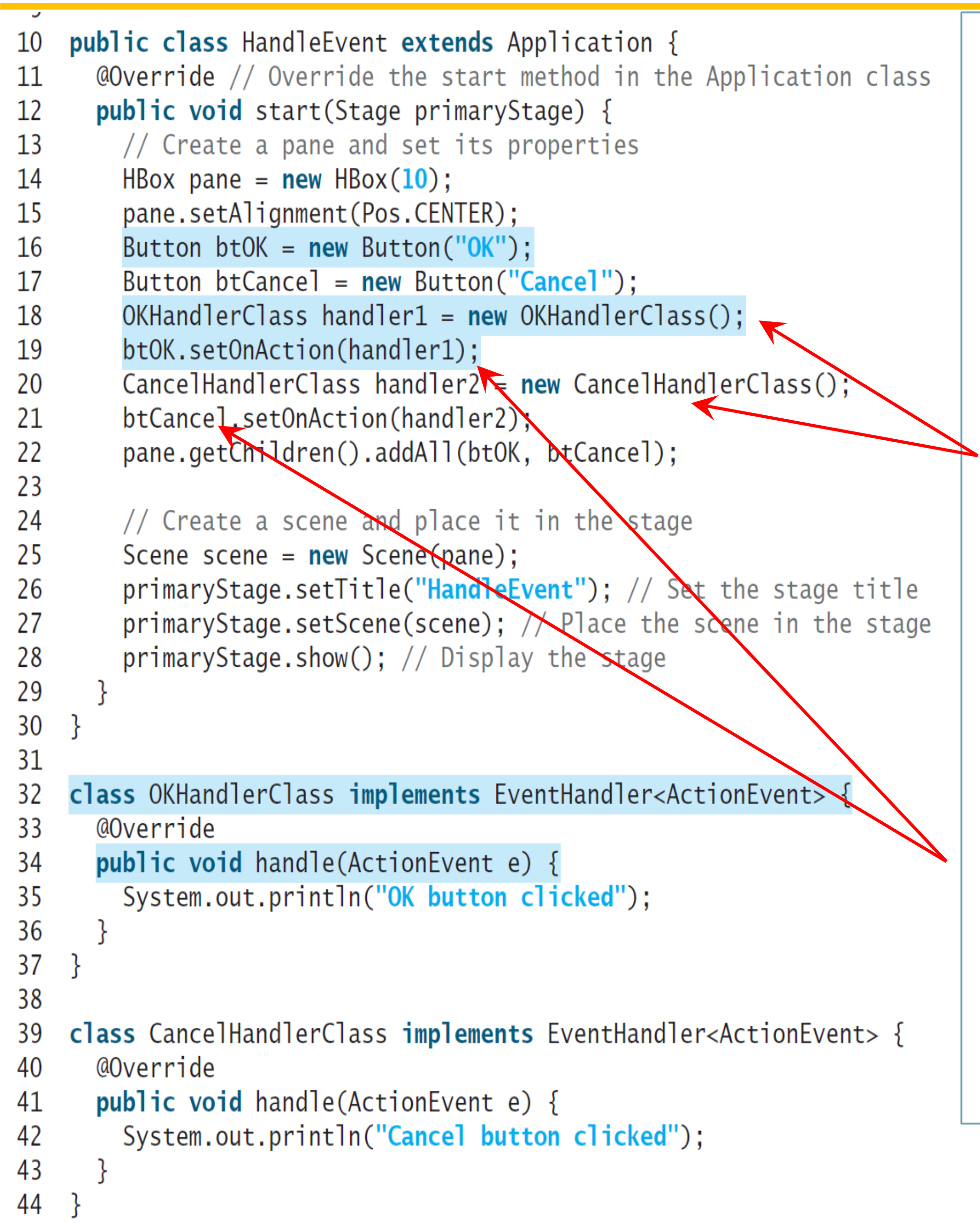
```
public class HandleEvent extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    HBox pane = new HBox(10);
    pane.setAlignment(Pos.CENTER);
    Button btOK = new Button("OK");
    Button btCancel = new Button("Cancel");
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    pane.getChildren().addAll(btOK, btCancel);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("HandleEvent"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}

class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}

class CancelHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("Cancel button clicked");
  }
}
```

These two handler classes' `handle` methods implement the functionality we want executed when the (click) event occurs

The first one announces (on the console) that the “OK” button was clicked

The second one announces (also on the console) that the “Cancel” button was clicked

LISTING 15.1 HandleEvent



```java
public class HandleEvent extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    HBox pane = new HBox(10);
    pane.setAlignment(Pos.CENTER);
    Button btOK = new Button("OK");
    Button btCancel = new Button("Cancel");
    OKHandlerClass handler1 = new OKHandlerClass();
    btOK.setOnAction(handler1);
    CancelHandlerClass handler2 = new CancelHandlerClass();
    btCancel.setOnAction(handler2);
    pane.getChildren().addAll(btOK, btCancel);

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("HandleEvent"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}

class OKHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("OK button clicked");
  }
}

class CancelHandlerClass implements EventHandler<ActionEvent> {
  @Override
  public void handle(ActionEvent e) {
    System.out.println("Cancel button clicked");
  }
}
```

The connection between the UI components and their event handlers is accomplished by *registering* the handlers

First, the handlers have to *exist* before they can "answer the phone"

Second, we tell the `btOK` and `btCancel` buttons who to "call" when an event occurs (`handler1` and `handler2`, respectively)

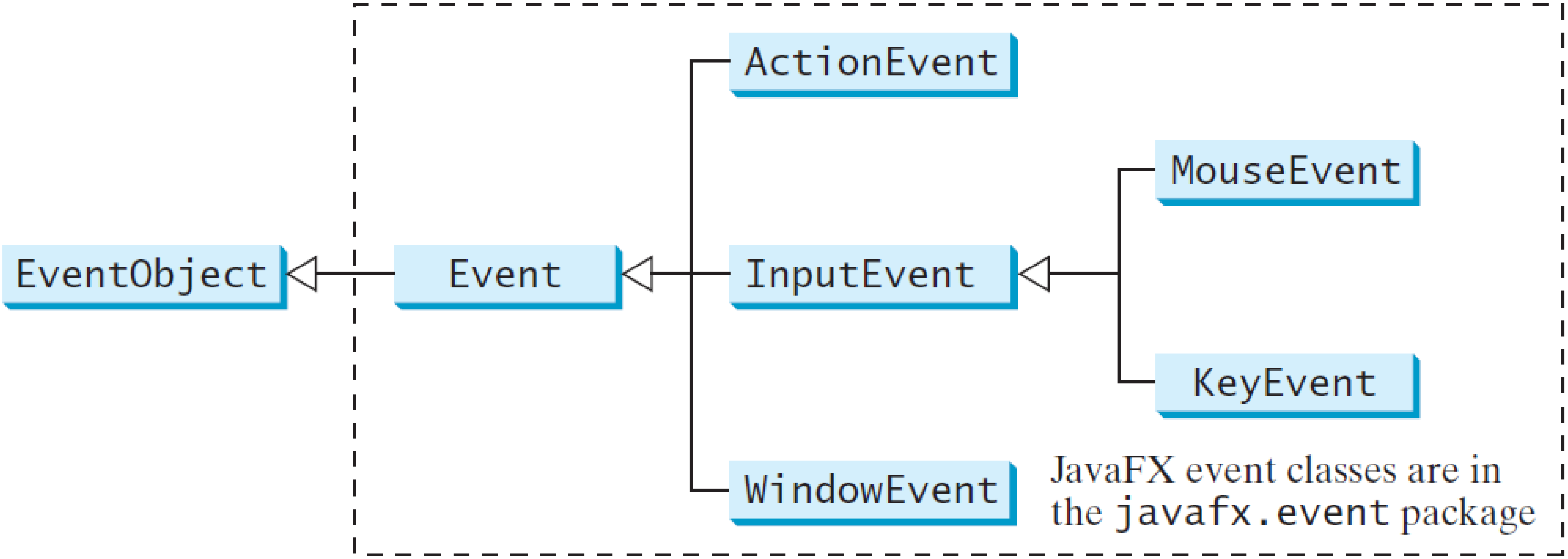
§15.2 Events and Event Sources

- *An event is an object created from an event source.*

  It is a signal (message) that something has happened
- Some events (like button clicks, key presses, mouse movements) are triggered by user action
- Some events (like timer ticks) can be generated by internal program activities
- In one sense, events are like exceptions:
- They're objects (in java) that correspond to something happening
  - An exception object is created (and *thrown*) when a problem occurs in our code (an exception). Who "gets" the exception object depends on whether we're in a try/catch block.
  - An event object is created (and *fired*) when something happens in the UI code. Who "gets" the event is whoever is registered as a handler for the object that generated the event (the *source*)

§15.2 Events and Event Sources

- `EventObject` hierarchy:



- `EventObject` has a `getSource()` method (so do its descendants), so an event handler can tell who generated an event it receives (caller ID)

§15.2 Events and Event Sources

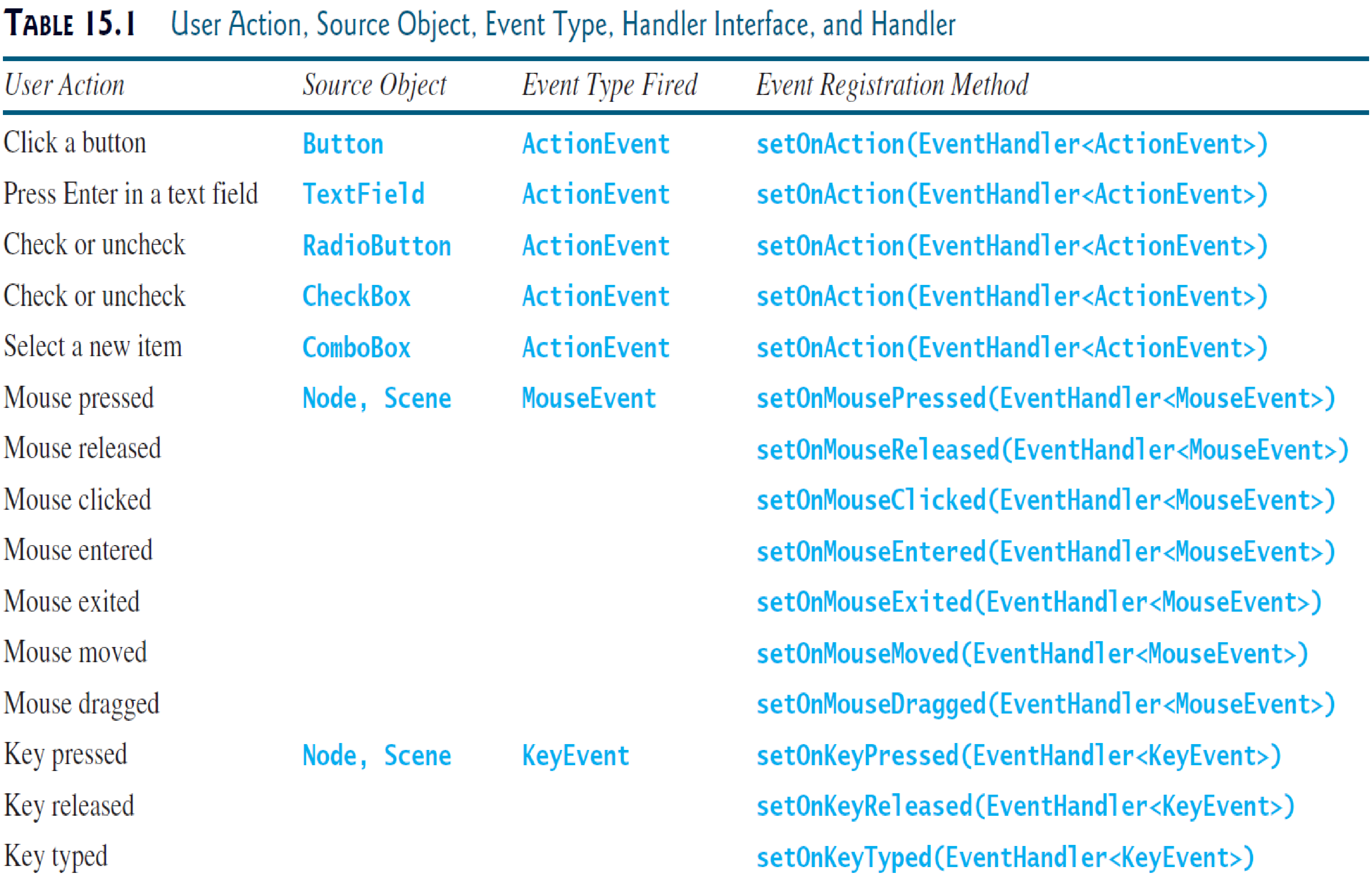


TABLE 15.1 User Action, Source Object, Event Type, Handler Interface, and Handler

User Action	*Source Object*	*Event Type Fired*	*Event Registration Method*
Click a button	`Button`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Press Enter in a text field	`TextField`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`RadioButton`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Check or uncheck	`CheckBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Select a new item	`ComboBox`	`ActionEvent`	`setOnAction(EventHandler<ActionEvent>)`
Mouse pressed	`Node, Scene`	`MouseEvent`	`setOnMousePressed(EventHandler<MouseEvent>)`
Mouse released			`setOnMouseReleased(EventHandler<MouseEvent>)`
Mouse clicked			`setOnMouseClicked(EventHandler<MouseEvent>)`
Mouse entered			`setOnMouseEntered(EventHandler<MouseEvent>)`
Mouse exited			`setOnMouseExited(EventHandler<MouseEvent>)`
Mouse moved			`setOnMouseMoved(EventHandler<MouseEvent>)`
Mouse dragged			`setOnMouseDragged(EventHandler<MouseEvent>)`
Key pressed	`Node, Scene`	`KeyEvent`	`setOnKeyPressed(EventHandler<KeyEvent>)`
Key released			`setOnKeyReleased(EventHandler<KeyEvent>)`
Key typed			`setOnKeyTyped(EventHandler<KeyEvent>)`

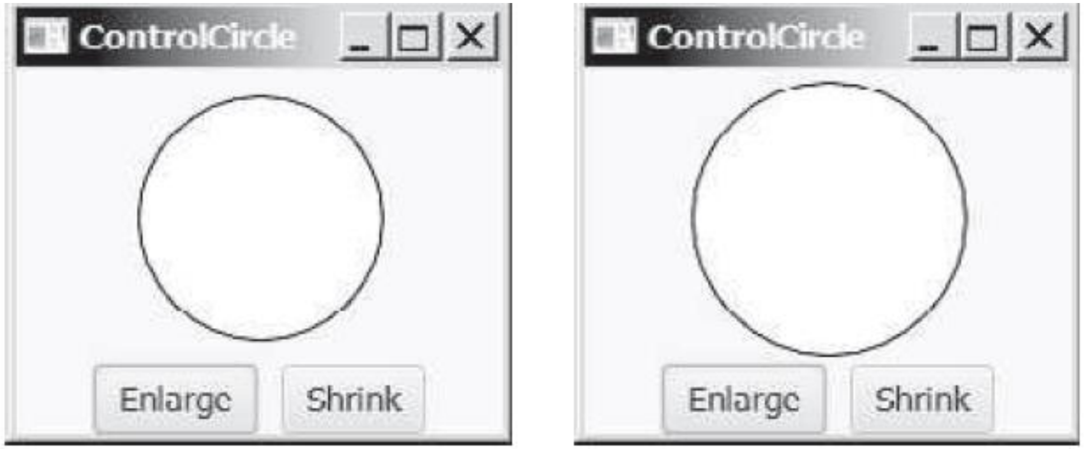
§15.2 Events and Event Sources

- What's most important for you to take away from this table is that different KINDS of source objects (GUI elements) generate different KINDS of events (e.g., buttons get *clicked*; not *moved*, like the mouse), and different kinds of events require different kinds of handlers.

- Although they both answer the phone and then provide some service, if the event is "Somebody is robbing the bank", I want a "police" KIND of handler, rather than a "customer service" KIND handler.

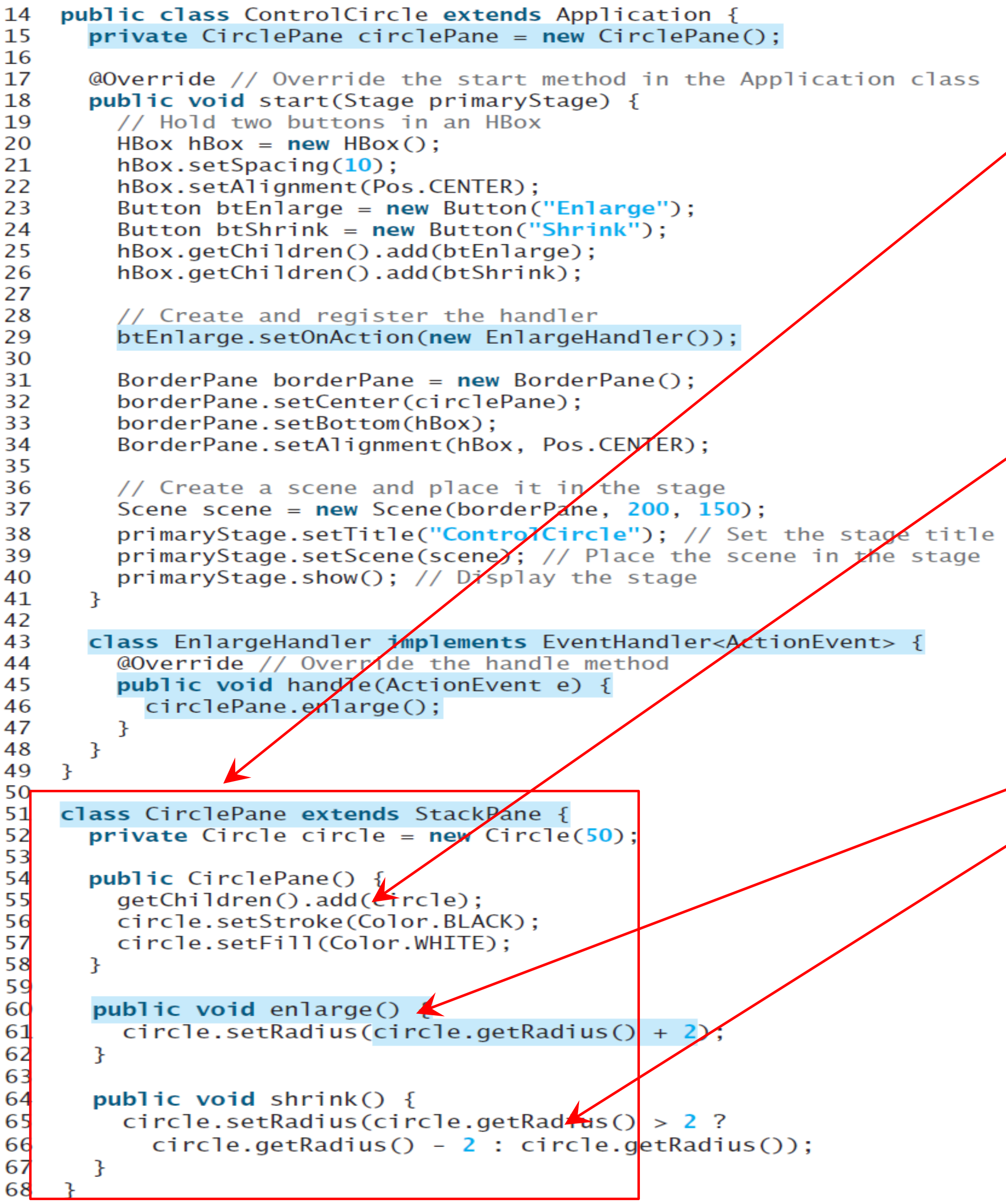
§15.3: LISTING 15.2 Handling Events

- The text walks through the incremental build of a program to display a circle, and a GUI with two buttons – one to enlarge the circle, and one to shrink the circle (Fig. 15.6):



- First, we write the code to create our GUI by extending `Application`, adding a `Pane`, and adding the (inert for now) `Buttons` to the UI:

§15.3: LISTING 15.2 Handling Events



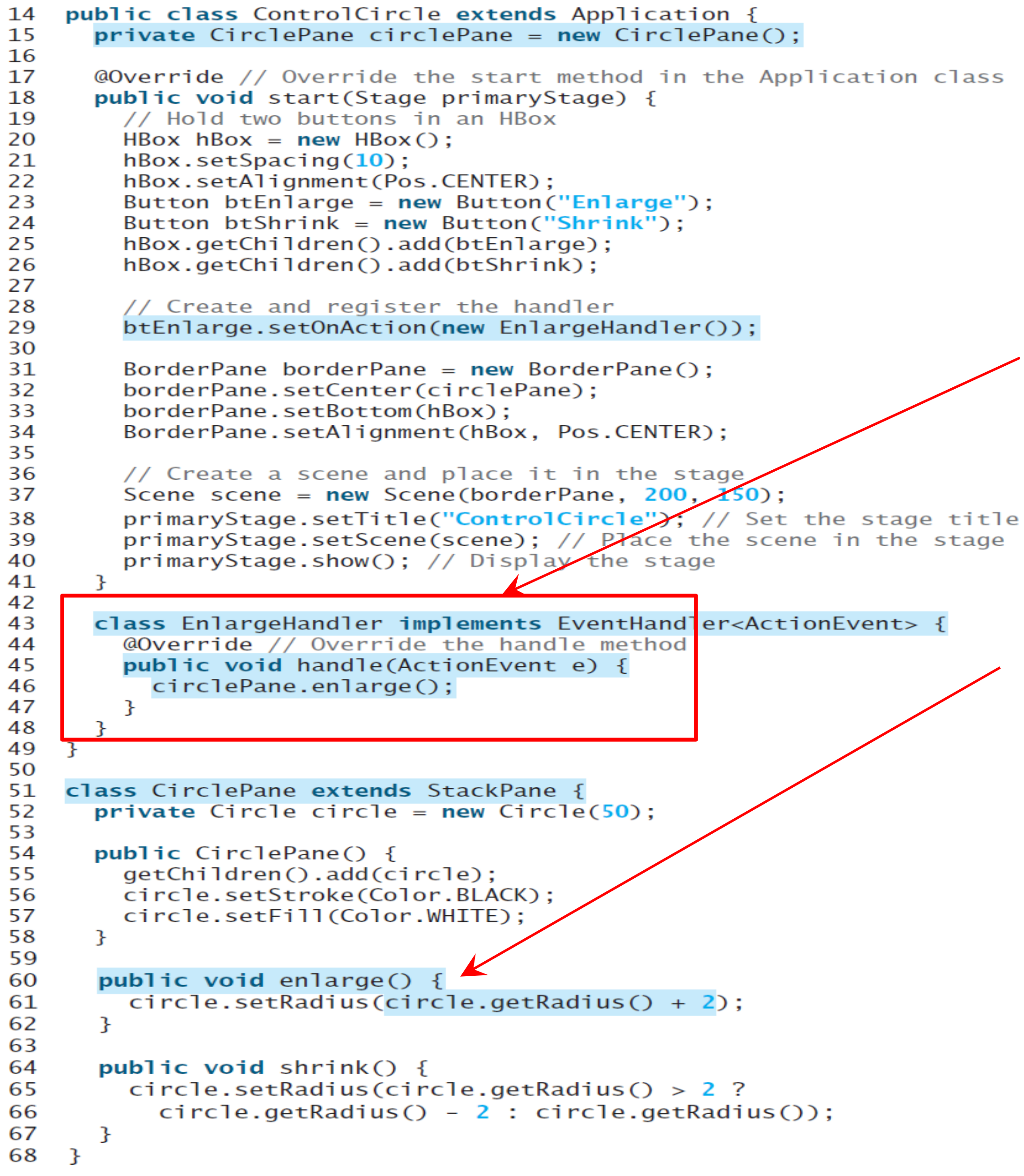
```
public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(new EnlargeHandler());

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override // Override the handle method
    public void handle(ActionEvent e) {
      circlePane.enlarge();
    }
  }
}

class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
    getChildren().add(circle);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
  }

  public void enlarge() {
    circle.setRadius(circle.getRadius() + 2);
  }

  public void shrink() {
    circle.setRadius(circle.getRadius() > 2 ?
      circle.getRadius() - 2 : circle.getRadius());
  }
}
```

The new `CirclePane` class will be an extension of a `StackPane` that contains the circle

Because it’s an extension of the `StackPane`, the `CirclePane` can access its own list of children, and add a circle in its constructor

The `CirclePane`’s `enlarge` and `shrink` methods simply add or subtract 2 to the circle’s current radius.

§15.3: LISTING 15.2 Handling Events



```
public class ControlCircle extends Application {
  private CirclePane circlePane = new CirclePane();

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Hold two buttons in an HBox
    HBox hBox = new HBox();
    hBox.setSpacing(10);
    hBox.setAlignment(Pos.CENTER);
    Button btEnlarge = new Button("Enlarge");
    Button btShrink = new Button("Shrink");
    hBox.getChildren().add(btEnlarge);
    hBox.getChildren().add(btShrink);

    // Create and register the handler
    btEnlarge.setOnAction(new EnlargeHandler());

    BorderPane borderPane = new BorderPane();
    borderPane.setCenter(circlePane);
    borderPane.setBottom(hBox);
    BorderPane.setAlignment(hBox, Pos.CENTER);

    // Create a scene and place it in the stage
    Scene scene = new Scene(borderPane, 200, 150);
    primaryStage.setTitle("ControlCircle"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  class EnlargeHandler implements EventHandler<ActionEvent> {
    @Override // Override the handle method
    public void handle(ActionEvent e) {
      circlePane.enlarge();
    }
  }
}

class CirclePane extends StackPane {
  private Circle circle = new Circle(50);

  public CirclePane() {
    getChildren().add(circle);
    circle.setStroke(Color.BLACK);
    circle.setFill(Color.WHITE);
  }

  public void enlarge() {
    circle.setRadius(circle.getRadius() + 2);
  }

  public void shrink() {
    circle.setRadius(circle.getRadius() > 2 ?
      circle.getRadius() - 2 : circle.getRadius());
  }
}
```

We know that the Buttons will need a class they can fire their events to.

We create the EnlargeHandler class to handle the ActionEvents that will come from the “Enlarge” Button.

The EnlargeHandler class has a handle method, which will only have to call the CirclePane’s enlarge method.

This code, as-written, doesn’t have any provision for the shrink button. The book leaves that as an exercise.

§15.4 Inner Classes

- An *inner class* is a class defined *within* the scope of another class
  - They are sometimes called *nested classes*.
- Inner classes are useful for defining listener (event-handling) classes
- The class within which the inner class is defined is called, *outer class*
- Normally, you define an inner class when it will be used by only the outer class
- Inner classes can help make your source files easier to manage

```
public class Test
{
    …
}
```

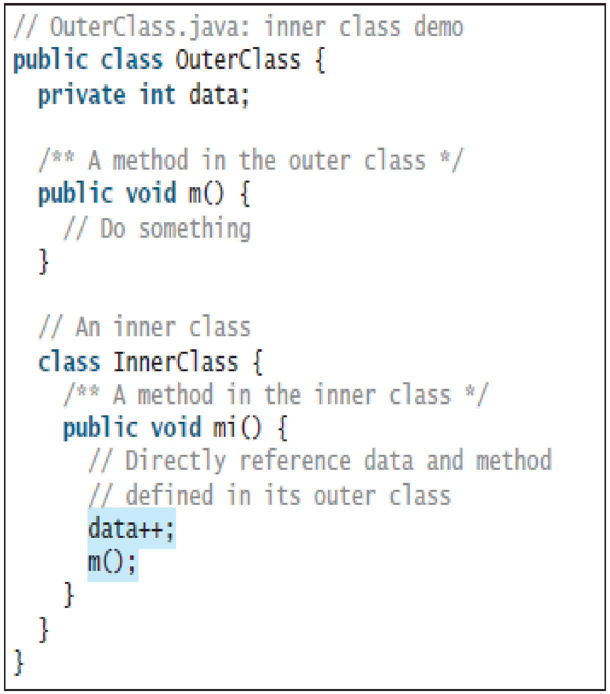
Test.java → Test.class

```
public class A
{
    …
}
```

A.java → A.class

```
public class Test
{
    …
    // inner class
    class A
    {
        …
    }
}
```

Test.java → Test.class
and → Test$A.class



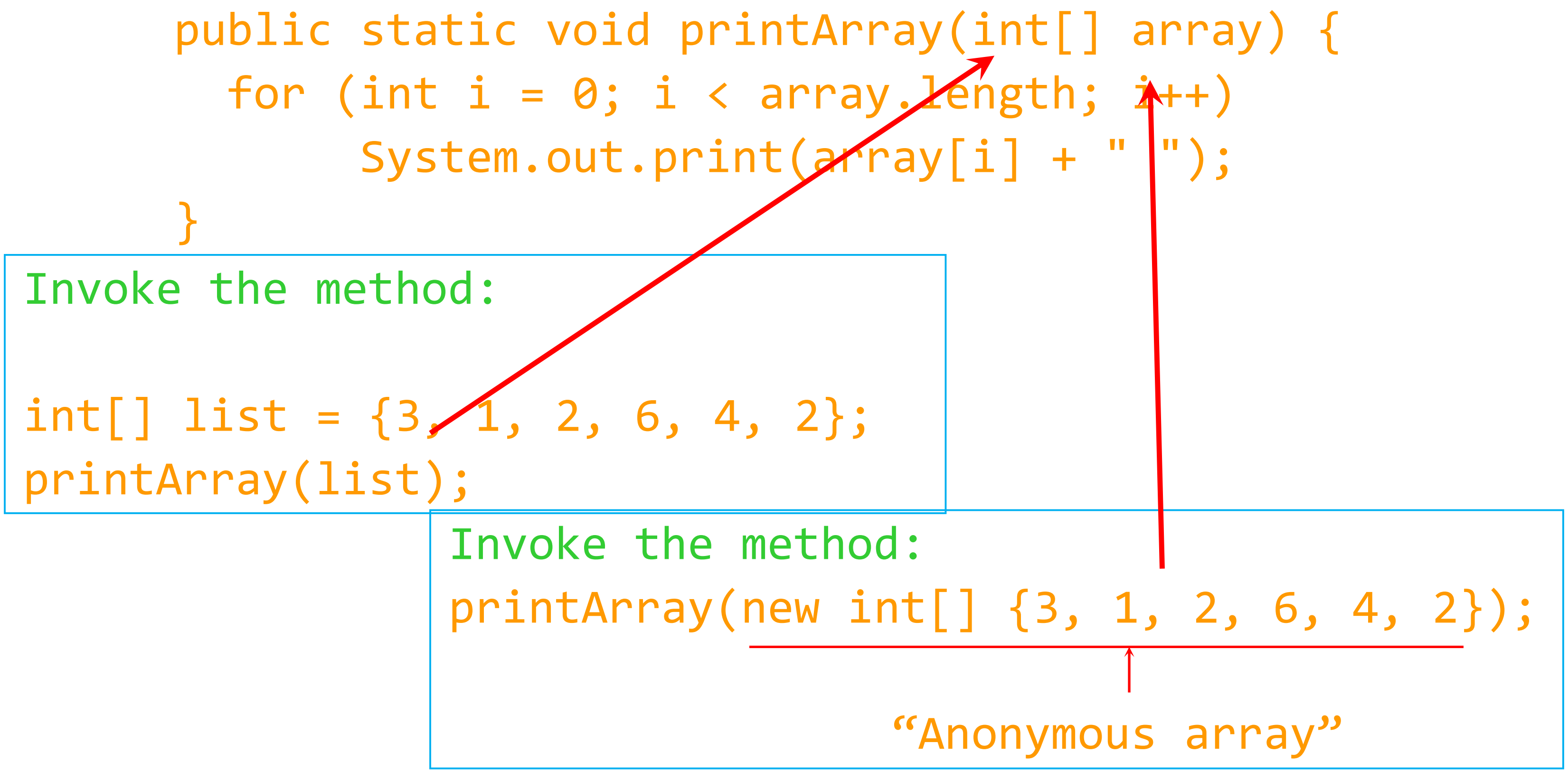
```
// OuterClass.java: inner class demo
public class OuterClass {
  private int data;

  /** A method in the outer class */
  public void m() {
    // Do something
  }

  // An inner class
  class InnerClass {
    /** A method in the inner class */
    public void mi() {
      // Directly reference data and method
      // defined in its outer class
      data++;
      m();
    }
  }
}
```

§15.4 Inner Classes

- The inner class can reference fields and methods of the outer class directly, so you don’t need to pass references from the outer class to the constructor of the inner class
- “A listener class is designed specifically to create a handler object for a GUI component (e.g., a button). The handler class will not be shared by other applications and therefore is appropriate to be defined inside the main class as an inner class.”
- An inner class can be defined with a visibility modifier subject to the same visibility rules applied to a member of the class.
- An inner class can be defined as **static**. A **static** inner class can be accessed using the outer class name. A **static** inner class cannot access nonstatic members of the outer class.
- We will be using inner classes to create handlers for UI elements in the outer class.

§15.5 Anonymous Inner Classes

- An anonymous inner class is an inner class without a name.
- It combines defining an inner class and creating an instance of the class into one step.
- An example of creating an anonymous array to pass to a method. which perform an instantiation and pass the reference to the method, rather than storing it in a reference variable and passing that.



```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++)
       System.out.print(array[i] + " ");
}
```

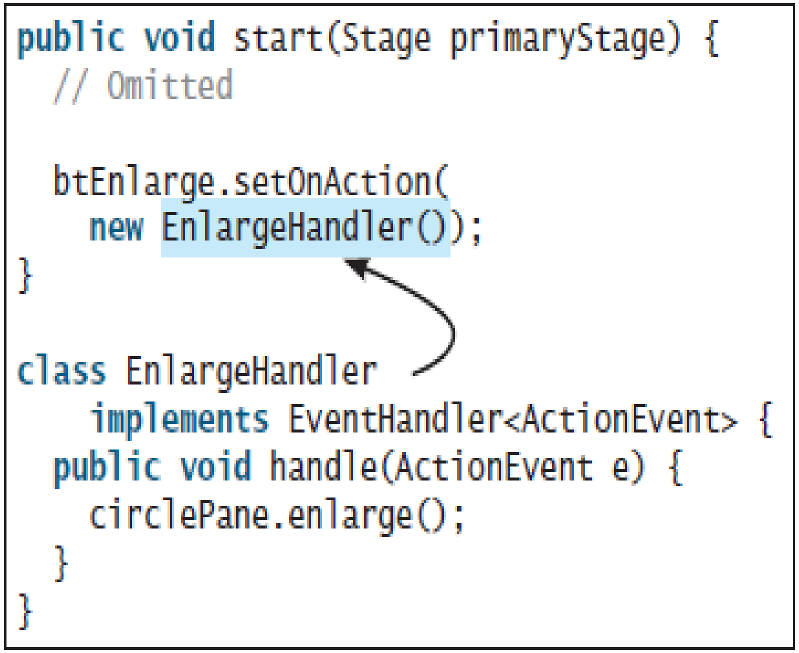
```
Invoke the method:

int[] list = {3, 1, 2, 6, 4, 2};
printArray(list);
```

```
Invoke the method:
printArray(new int[] {3, 1, 2, 6, 4, 2});
```

“Anonymous array”

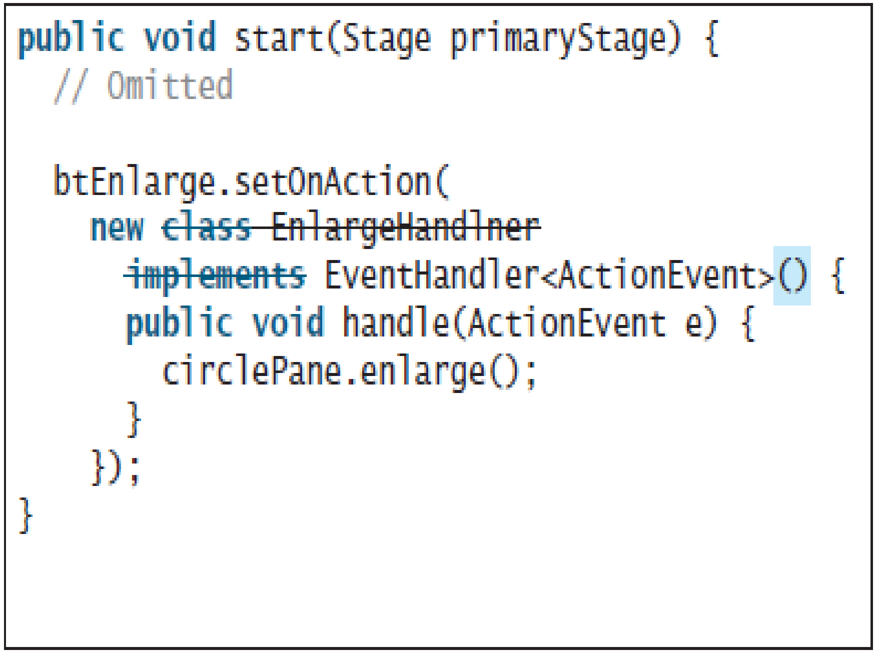
§15.5 Anonymous Inner Classes

- An anonymous inner class combines a class’s definition and instantiation in one step:
- The syntax for an anonymous inner class is shown below
  new SuperClassName/InterfaceName() {
    // Implement or override methods in superclass or interface
  // Other methods if necessary }



```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

class EnlargeHandler
    implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    circlePane.enlarge();
  }
}
```

(a) Inner class EnlargeListener



```
public void start(Stage primaryStage) {
  // Omitted

  btEnlarge.setOnAction(
    new ~~class EnlargeHandlner~~
      ~~implements~~ EventHandler<ActionEvent>() {
      public void handle(ActionEvent e) {
        circlePane.enlarge();
      }
    });
}
```

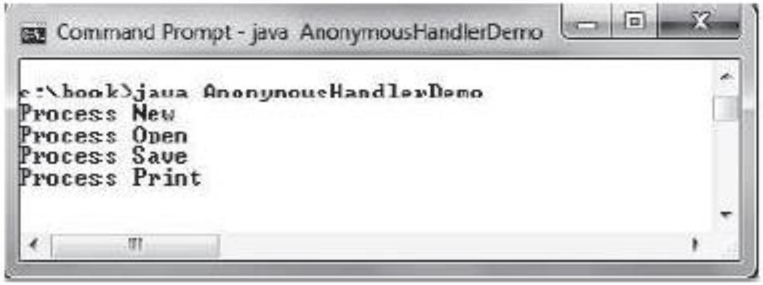
(b) Anonymous inner class

§15.5 Anonymous Inner Classes

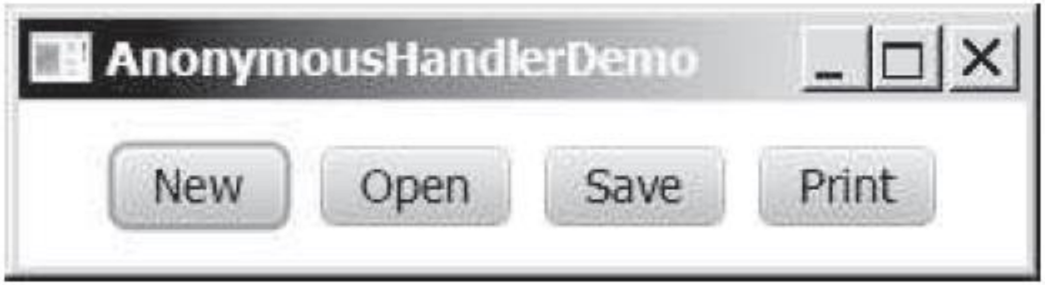
- Since an anonymous inner class is a special kind of inner class, it is treated like an inner class with the following four features:
  1. An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit extends or implements clause.
  2. An anonymous inner class must implement all the abstract methods in the superclass or in the interface (it must "go concrete").
  3. An anonymous inner class always uses the no-arg constructor from its superclass to create an instance. If an anonymous inner class implements an interface, the constructor is Object().
  4. An anonymous inner class is compiled into a class named OuterClassName$n.class. For example, if the outer class Test has two anonymous inner classes, they are compiled into Test$1.class and Test$2.class.

§15.5 Anonymous Inner Classes

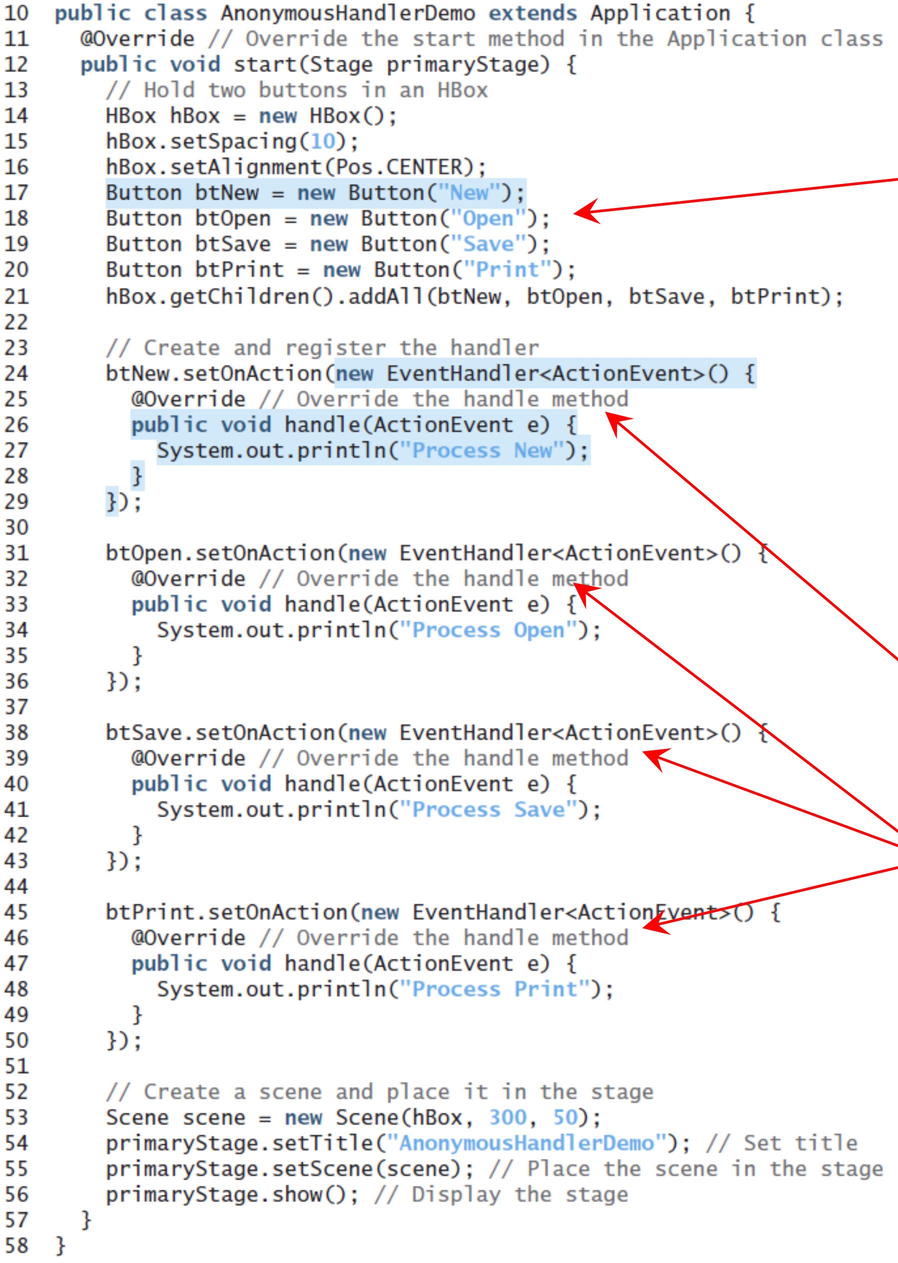
- In Listing 15.4 (pp. 595-596, and next slide), Liang shows the code for an application with four buttons.
- When each button is clicked, it produces output on the console (although we can certainly have its click event handler do anything we want):



```
c:\book>java AnonymousHandlerDemo
Process New
Process Open
Process Save
Process Print
```



§15.5 Anonymous Inner Classes



```
10  public class AnonymousHandlerDemo extends Application {
11    @Override // Override the start method in the Application class
12    public void start(Stage primaryStage) {
13      // Hold two buttons in an HBox
14      HBox hBox = new HBox();
15      hBox.setSpacing(10);
16      hBox.setAlignment(Pos.CENTER);
17      Button btNew = new Button("New");
18      Button btOpen = new Button("Open");
19      Button btSave = new Button("Save");
20      Button btPrint = new Button("Print");
21      hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);
22
23      // Create and register the handler
24      btNew.setOnAction(new EventHandler<ActionEvent>() {
25        @Override // Override the handle method
26        public void handle(ActionEvent e) {
27          System.out.println("Process New");
28        }
29      });
30
31      btOpen.setOnAction(new EventHandler<ActionEvent>() {
32        @Override // Override the handle method
33        public void handle(ActionEvent e) {
34          System.out.println("Process Open");
35        }
36      });
37
38      btSave.setOnAction(new EventHandler<ActionEvent>() {
39        @Override // Override the handle method
40        public void handle(ActionEvent e) {
41          System.out.println("Process Save");
42        }
43      });
44
45      btPrint.setOnAction(new EventHandler<ActionEvent>() {
46        @Override // Override the handle method
47        public void handle(ActionEvent e) {
48          System.out.println("Process Print");
49        }
50      });
51
52      // Create a scene and place it in the stage
53      Scene scene = new Scene(hBox, 300, 50);
54      primaryStage.setTitle("AnonymousHandlerDemo"); // Set title
55      primaryStage.setScene(scene); // Place the scene in the stage
56      primaryStage.show(); // Display the stage
57    }
58  }
```

I've omitted the `import` statements from the top

Lines 14 – 21 create an `HBox` and four `Buttons`, which it adds to the `HBox`.

The "New" button (`btnNew`) needs to be registered with the object that will receive (and handle) its `ActionEvents`.

That object will have to be an instance of a class that `inplements EventHanlder<ActionEvent>`

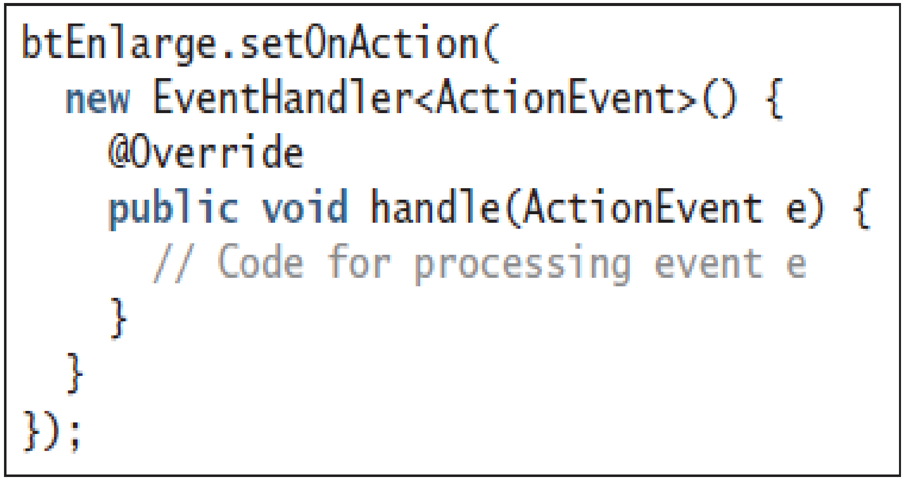
That's precisely what the blue-shaded code in lines 24 – 29 IS

The same format is repeated to register an anonymous inner `EventHandler<ActionEvent>` class for `btnOpen`, `btnSave`, and `btnPrint`

By "embedding" the handler classes inside the class with the UI, we eliminate the need for several explicitly-defined classes that will be used only to handle events for one object!

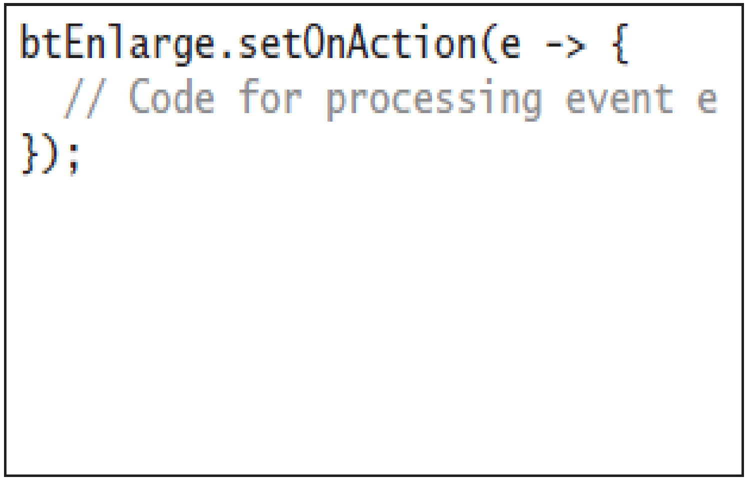
§15.6 Lambda Expressions and Events

- Lambda Expressions (new to Java 8) can be considered an anonymous inner class with an abbreviated syntax.
- The compiler treats lambda expressions like an object created from an anonymous inner class
- The compiler knows that the parameter of the setOnAction method must be something of type EventHandler<ActionEvent>, and that particular interface has only one abstract method, so the code between the braces *must* be the statements for that method



```
btEnlarge.setOnAction(
  new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent e) {
      // Code for processing event e
    }
  }
});
```

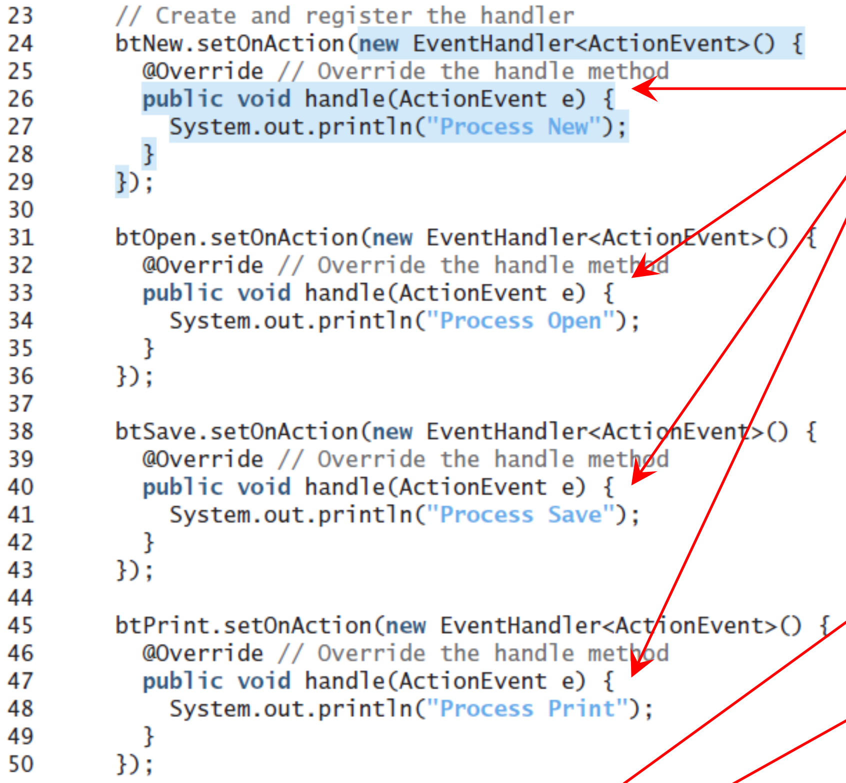
(a) Anonymous inner class event handler



```
btEnlarge.setOnAction(e -> {
  // Code for processing event e
});
```

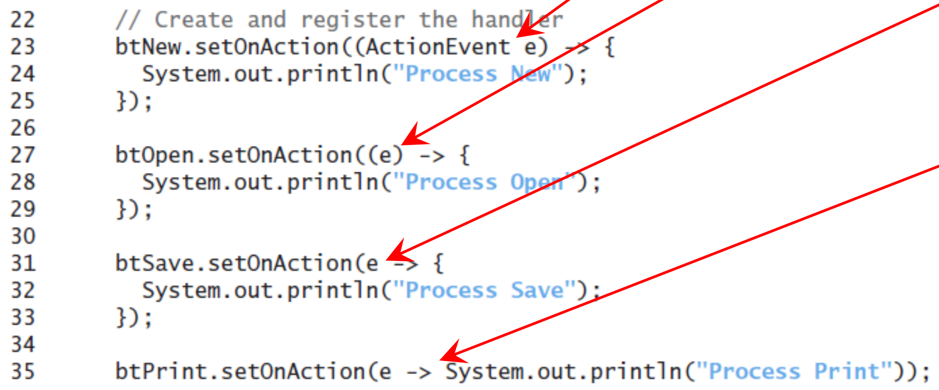
(b) Lambda expression event handler

§15.6 Lambda Expressions and Events



```
// Create and register the handler
btNew.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process New");
  }
});

btOpen.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process Open");
  }
});

btSave.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process Save");
  }
});

btPrint.setOnAction(new EventHandler<ActionEvent>() {
  @Override // Override the handle method
  public void handle(ActionEvent e) {
    System.out.println("Process Print");
  }
});
```

In the original code, each button had its own complete anonymous inner class to handle the `ActionEvent` from its corresponding button



```
// Create and register the handler
btNew.setOnAction((ActionEvent e) -> {
  System.out.println("Process New");
});

btOpen.setOnAction((e) -> {
  System.out.println("Process Open");
});

btSave.setOnAction(e -> {
  System.out.println("Process Save");
});

btPrint.setOnAction(e -> System.out.println("Process Print"));
```

In the updated version, we show four different ways (styles) of expressing the registration of the event handler using lambda expressions

The first explicitly gives the type of e

The second omits the type, because the compiler can infer that e is of type `ActionEvent`

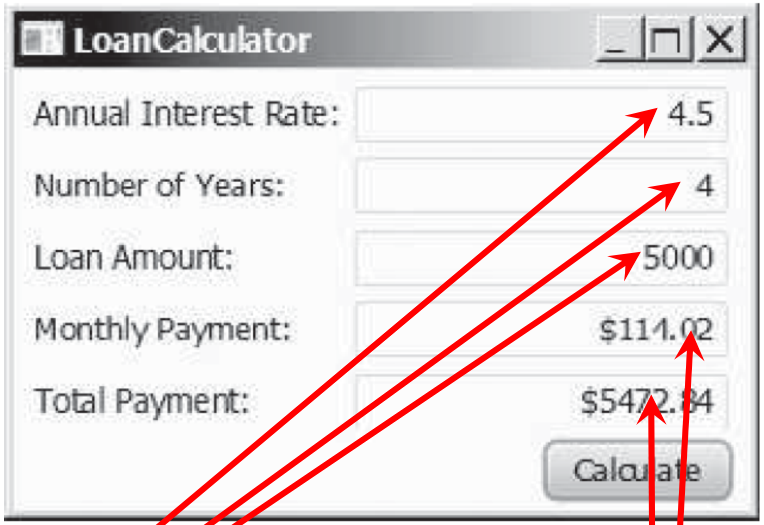
The third omits the parentheses, because there is only one parameter

The fourth omits the braces, because there is only one line of code (much like a one-line then or else clause of an `if` / `then` / `else`, or a one-statement loop body)

Lambda expressions make for cleaner code!

§15.7: Case Study: Loan Calculator

- This chapter began with:
- Suppose you want to develop a GUI application to calculate loan payments:



Annual Interest Rate: 4.5

Number of Years: 4

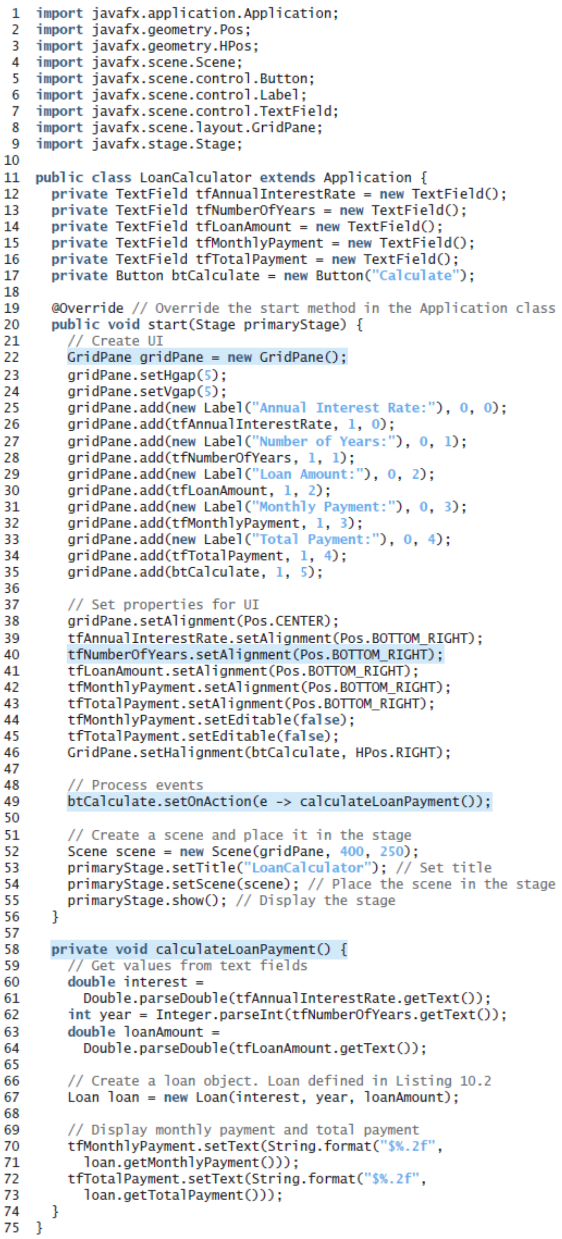
Loan Amount: 5000

Monthly Payment: $114.02

Total Payment: $5472.84

- This section builds this program, start-to-finish
- We introduce the `TextField` node – a box into which we can type text
- The Monthly Payment and Total Payment fields are <u>*display-only*</u> (the user can’t type In these)
- Let’s take a closer look at this code…

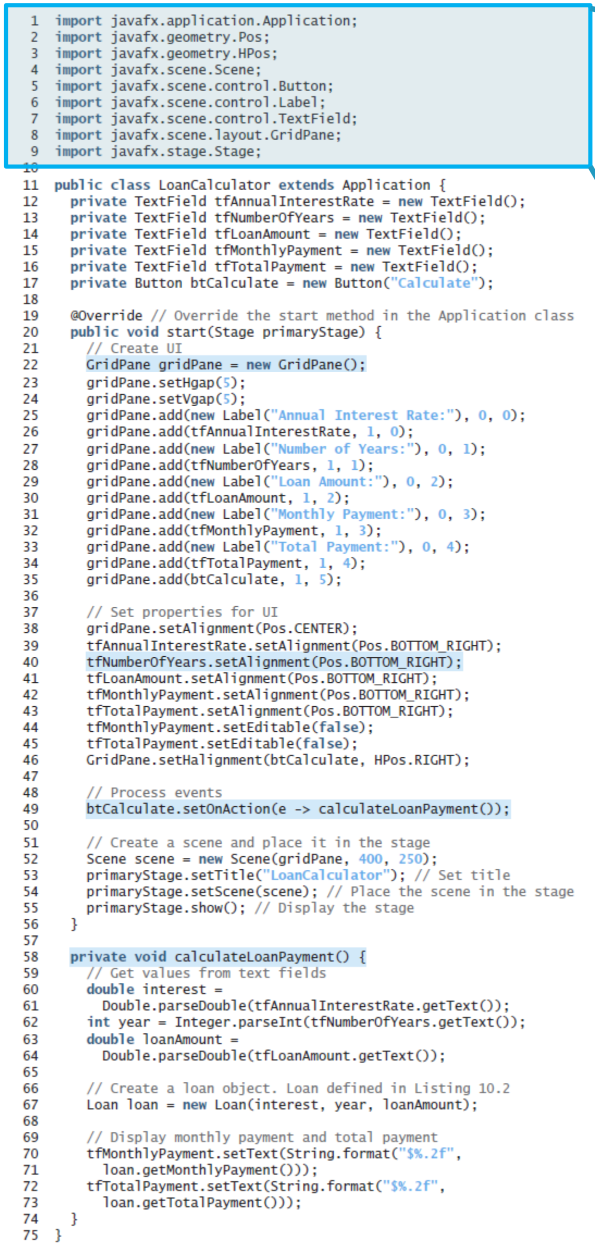
§15.7: Case Study: Loan Calculator



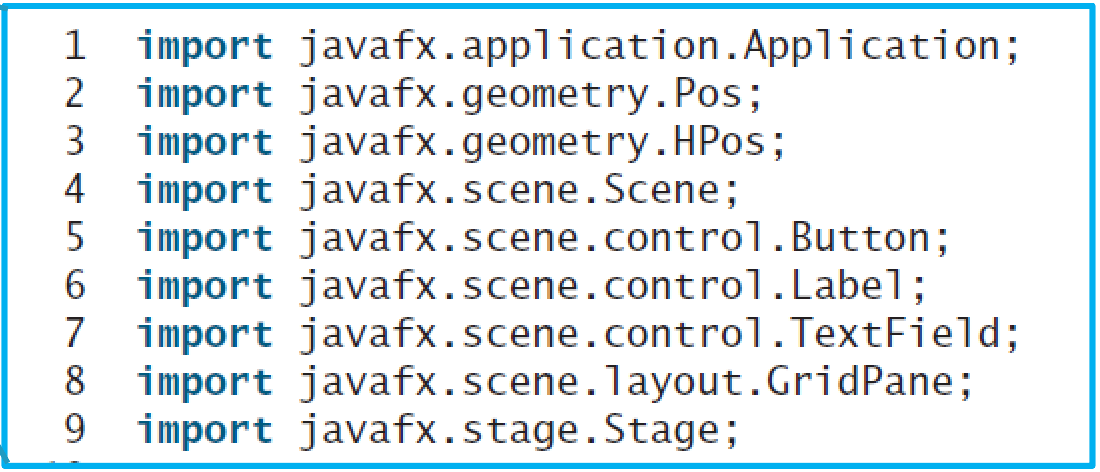
```java
import javafx.application.Application;
import javafx.geometry.Pos;
import javafx.geometry.HPos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.stage.Stage;

public class LoanCalculator extends Application {
  private TextField tfAnnualInterestRate = new TextField();
  private TextField tfNumberOfYears = new TextField();
  private TextField tfLoanAmount = new TextField();
  private TextField tfMonthlyPayment = new TextField();
  private TextField tfTotalPayment = new TextField();
  private Button btCalculate = new Button("Calculate");

  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create UI
    GridPane gridPane = new GridPane();
    gridPane.setHgap(5);
    gridPane.setVgap(5);
    gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
    gridPane.add(tfAnnualInterestRate, 1, 0);
    gridPane.add(new Label("Number of Years:"), 0, 1);
    gridPane.add(tfNumberOfYears, 1, 1);
    gridPane.add(new Label("Loan Amount:"), 0, 2);
    gridPane.add(tfLoanAmount, 1, 2);
    gridPane.add(new Label("Monthly Payment:"), 0, 3);
    gridPane.add(tfMonthlyPayment, 1, 3);
    gridPane.add(new Label("Total Payment:"), 0, 4);
    gridPane.add(tfTotalPayment, 1, 4);
    gridPane.add(btCalculate, 1, 5);

    // Set properties for UI
    gridPane.setAlignment(Pos.CENTER);
    tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
    tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
    tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
    tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
    tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
    tfMonthlyPayment.setEditable(false);
    tfTotalPayment.setEditable(false);
    GridPane.setHalignment(btCalculate, HPos.RIGHT);

    // Process events
    btCalculate.setOnAction(e -> calculateLoanPayment());

    // Create a scene and place it in the stage
    Scene scene = new Scene(gridPane, 400, 250);
    primaryStage.setTitle("LoanCalculator"); // Set title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  private void calculateLoanPayment() {
    // Get values from text fields
    double interest =
      Double.parseDouble(tfAnnualInterestRate.getText());
    int year = Integer.parseInt(tfNumberOfYears.getText());
    double loanAmount =
      Double.parseDouble(tfLoanAmount.getText());

    // Create a loan object. Loan defined in Listing 10.2
    Loan loan = new Loan(interest, year, loanAmount);

    // Display monthly payment and total payment
    tfMonthlyPayment.setText(String.format("$%.2f",
      loan.getMonthlyPayment()));
    tfTotalPayment.setText(String.format("$%.2f",
      loan.getTotalPayment()));
  }
}
```

Seventy-five lines of code simply isn’t readable at this size, so we’ll go through this program in pieces

§15.7: Case Study: Loan Calculator



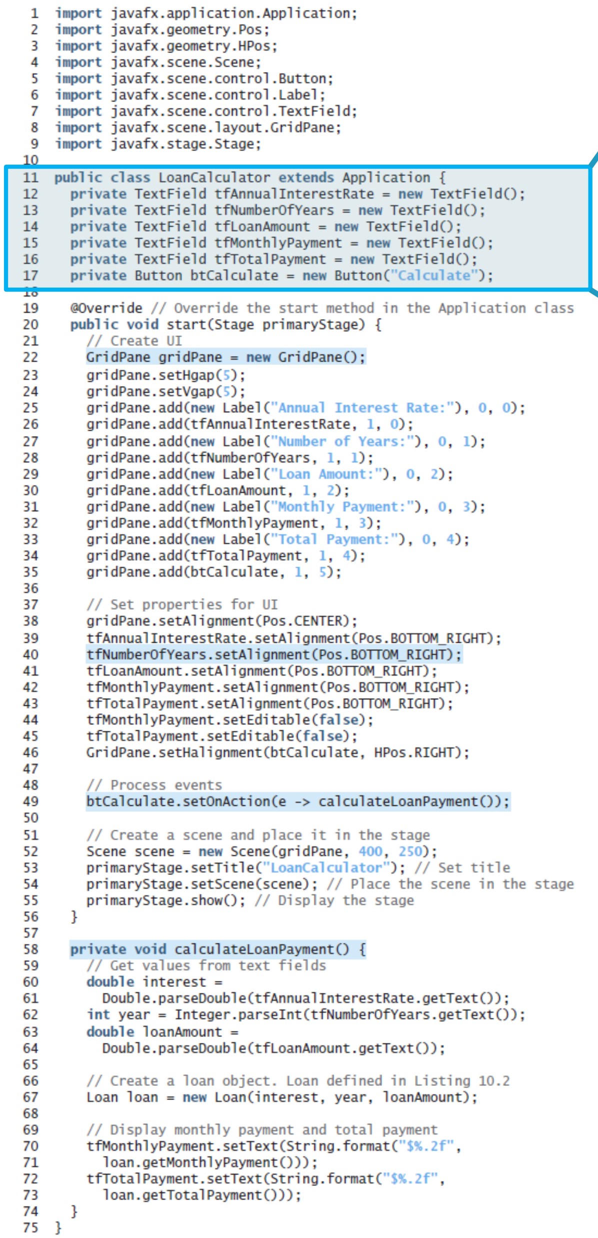
```
 1 import javafx.application.Application;
 2 import javafx.geometry.Pos;
 3 import javafx.geometry.HPos;
 4 import javafx.scene.Scene;
 5 import javafx.scene.control.Button;
 6 import javafx.scene.control.Label;
 7 import javafx.scene.control.TextField;
 8 import javafx.scene.layout.GridPane;
 9 import javafx.stage.Stage;
10
11 public class LoanCalculator extends Application {
12   private TextField tfAnnualInterestRate = new TextField();
13   private TextField tfNumberOfYears = new TextField();
14   private TextField tfLoanAmount = new TextField();
15   private TextField tfMonthlyPayment = new TextField();
16   private TextField tfTotalPayment = new TextField();
17   private Button btCalculate = new Button("Calculate");
18
19   @Override // Override the start method in the Application class
20   public void start(Stage primaryStage) {
21     // Create UI
22     GridPane gridPane = new GridPane();
23     gridPane.setHgap(5);
24     gridPane.setVgap(5);
25     gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26     gridPane.add(tfAnnualInterestRate, 1, 0);
27     gridPane.add(new Label("Number of Years:"), 0, 1);
28     gridPane.add(tfNumberOfYears, 1, 1);
29     gridPane.add(new Label("Loan Amount:"), 0, 2);
30     gridPane.add(tfLoanAmount, 1, 2);
31     gridPane.add(new Label("Monthly Payment:"), 0, 3);
32     gridPane.add(tfMonthlyPayment, 1, 3);
33     gridPane.add(new Label("Total Payment:"), 0, 4);
34     gridPane.add(tfTotalPayment, 1, 4);
35     gridPane.add(btCalculate, 1, 5);
36
37     // Set properties for UI
38     gridPane.setAlignment(Pos.CENTER);
39     tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40     tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41     tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42     tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43     tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44     tfMonthlyPayment.setEditable(false);
45     tfTotalPayment.setEditable(false);
46     GridPane.setHalignment(btCalculate, HPos.RIGHT);
47
48     // Process events
49     btCalculate.setOnAction(e -> calculateLoanPayment());
50
51     // Create a scene and place it in the stage
52     Scene scene = new Scene(gridPane, 400, 250);
53     primaryStage.setTitle("LoanCalculator"); // Set title
54     primaryStage.setScene(scene); // Place the scene in the stage
55     primaryStage.show(); // Display the stage
56   }
57
58   private void calculateLoanPayment() {
59     // Get values from text fields
60     double interest =
61       Double.parseDouble(tfAnnualInterestRate.getText());
62     int year = Integer.parseInt(tfNumberOfYears.getText());
63     double loanAmount =
64       Double.parseDouble(tfLoanAmount.getText());
65
66     // Create a loan object. Loan defined in Listing 10.2
67     Loan loan = new Loan(interest, year, loanAmount);
68
69     // Display monthly payment and total payment
70     tfMonthlyPayment.setText(String.format("$%.2f",
71       loan.getMonthlyPayment()));
72     tfTotalPayment.setText(String.format("$%.2f",
73       loan.getTotalPayment()));
74   }
75 }
```



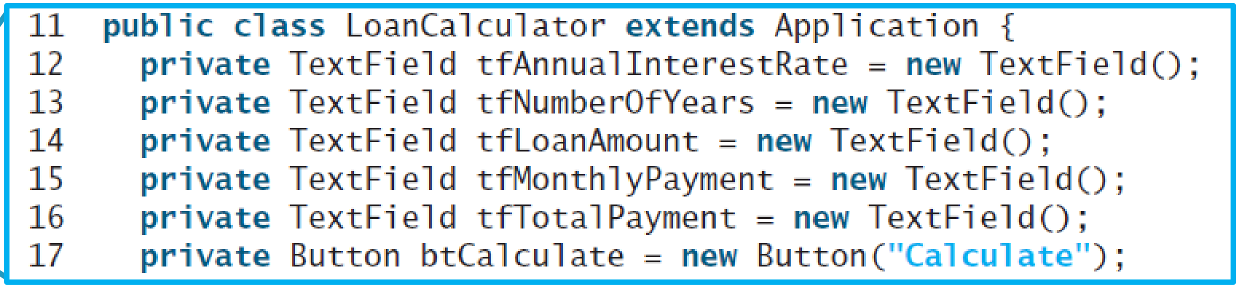
```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.geometry.HPos;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.Label;
7  import javafx.scene.control.TextField;
8  import javafx.scene.layout.GridPane;
9  import javafx.stage.Stage;
```

Lines 1 – 9 simply import the JavaFX components we will need for this program

§15.7: Case Study: Loan Calculator



```java
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.geometry.HPos;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.Label;
7  import javafx.scene.control.TextField;
8  import javafx.scene.layout.GridPane;
9  import javafx.stage.Stage;
10
11 public class LoanCalculator extends Application {
12   private TextField tfAnnualInterestRate = new TextField();
13   private TextField tfNumberOfYears = new TextField();
14   private TextField tfLoanAmount = new TextField();
15   private TextField tfMonthlyPayment = new TextField();
16   private TextField tfTotalPayment = new TextField();
17   private Button btCalculate = new Button("Calculate");
18
19   @Override // Override the start method in the Application class
20   public void start(Stage primaryStage) {
21     // Create UI
22     GridPane gridPane = new GridPane();
23     gridPane.setHgap(5);
24     gridPane.setVgap(5);
25     gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26     gridPane.add(tfAnnualInterestRate, 1, 0);
27     gridPane.add(new Label("Number of Years:"), 0, 1);
28     gridPane.add(tfNumberOfYears, 1, 1);
29     gridPane.add(new Label("Loan Amount:"), 0, 2);
30     gridPane.add(tfLoanAmount, 1, 2);
31     gridPane.add(new Label("Monthly Payment:"), 0, 3);
32     gridPane.add(tfMonthlyPayment, 1, 3);
33     gridPane.add(new Label("Total Payment:"), 0, 4);
34     gridPane.add(tfTotalPayment, 1, 4);
35     gridPane.add(btCalculate, 1, 5);
36
37     // Set properties for UI
38     gridPane.setAlignment(Pos.CENTER);
39     tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40     tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41     tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42     tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43     tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44     tfMonthlyPayment.setEditable(false);
45     tfTotalPayment.setEditable(false);
46     GridPane.setHalignment(btCalculate, HPos.RIGHT);
47
48     // Process events
49     btCalculate.setOnAction(e -> calculateLoanPayment());
50
51     // Create a scene and place it in the stage
52     Scene scene = new Scene(gridPane, 400, 250);
53     primaryStage.setTitle("LoanCalculator"); // Set title
54     primaryStage.setScene(scene); // Place the scene in the stage
55     primaryStage.show(); // Display the stage
56   }
57
58   private void calculateLoanPayment() {
59     // Get values from text fields
60     double interest =
61       Double.parseDouble(tfAnnualInterestRate.getText());
62     int year = Integer.parseInt(tfNumberOfYears.getText());
63     double loanAmount =
64       Double.parseDouble(tfLoanAmount.getText());
65
66     // Create a loan object. Loan defined in Listing 10.2
67     Loan loan = new Loan(interest, year, loanAmount);
68
69     // Display monthly payment and total payment
70     tfMonthlyPayment.setText(String.format("$%.2f",
71       loan.getMonthlyPayment()));
72     tfTotalPayment.setText(String.format("$%.2f",
73       loan.getTotalPayment()));
74   }
75 }
```

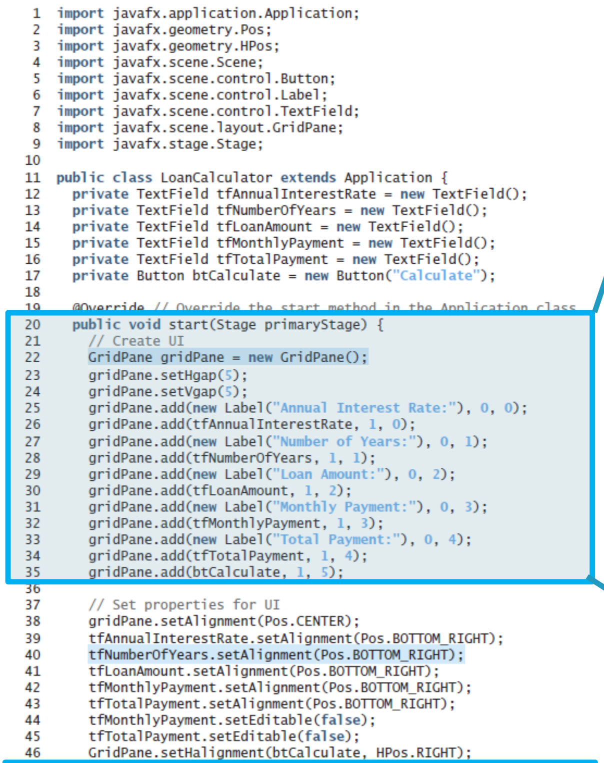


```java
11 public class LoanCalculator extends Application {
12   private TextField tfAnnualInterestRate = new TextField();
13   private TextField tfNumberOfYears = new TextField();
14   private TextField tfLoanAmount = new TextField();
15   private TextField tfMonthlyPayment = new TextField();
16   private TextField tfTotalPayment = new TextField();
17   private Button btCalculate = new Button("Calculate");
```

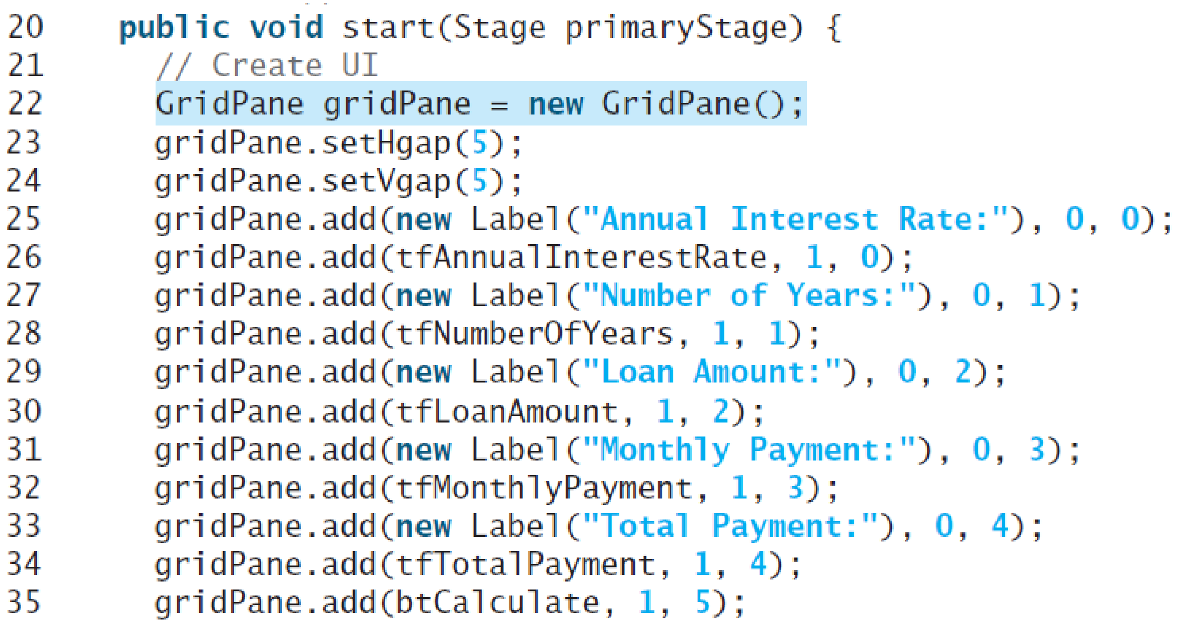
Lines 11 – 17 begin our `LoanCalculator` class (an extension of `Application`), and create the five `TextFields` (and the “Calculate” `Button`) as fields of the class

These are set up as fields so that the method to do the calculation (see below) can *see* them, and we don’t have to pass them all as parameters (by making them fields, they have class-wide scope)

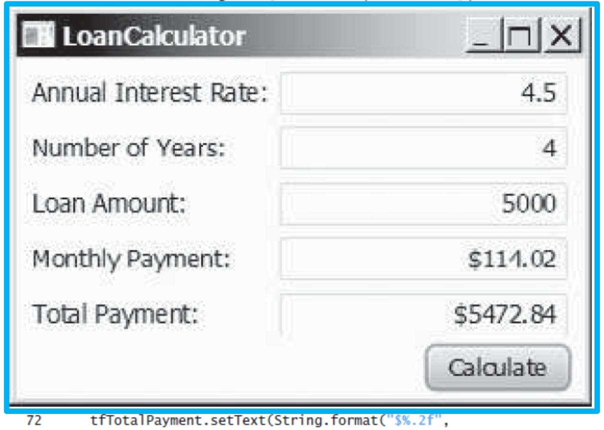
§15.7: Case Study: Loan Calculator



```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.geometry.HPos;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.Label;
7  import javafx.scene.control.TextField;
8  import javafx.scene.layout.GridPane;
9  import javafx.stage.Stage;
10
11 public class LoanCalculator extends Application {
12   private TextField tfAnnualInterestRate = new TextField();
13   private TextField tfNumberOfYears = new TextField();
14   private TextField tfLoanAmount = new TextField();
15   private TextField tfMonthlyPayment = new TextField();
16   private TextField tfTotalPayment = new TextField();
17   private Button btCalculate = new Button("Calculate");
18
19   @Override // Override the start method in the Application class
20   public void start(Stage primaryStage) {
21     // Create UI
22     GridPane gridPane = new GridPane();
23     gridPane.setHgap(5);
24     gridPane.setVgap(5);
25     gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26     gridPane.add(tfAnnualInterestRate, 1, 0);
27     gridPane.add(new Label("Number of Years:"), 0, 1);
28     gridPane.add(tfNumberOfYears, 1, 1);
29     gridPane.add(new Label("Loan Amount:"), 0, 2);
30     gridPane.add(tfLoanAmount, 1, 2);
31     gridPane.add(new Label("Monthly Payment:"), 0, 3);
32     gridPane.add(tfMonthlyPayment, 1, 3);
33     gridPane.add(new Label("Total Payment:"), 0, 4);
34     gridPane.add(tfTotalPayment, 1, 4);
35     gridPane.add(btCalculate, 1, 5);
36
37     // Set properties for UI
38     gridPane.setAlignment(Pos.CENTER);
39     tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40     tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41     tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42     tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43     tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44     tfMonthlyPayment.setEditable(false);
45     tfTotalPayment.setEditable(false);
46     GridPane.setHalignment(btCalculate, HPos.RIGHT);
```



```
20   public void start(Stage primaryStage) {
21     // Create UI
22     GridPane gridPane = new GridPane();
23     gridPane.setHgap(5);
24     gridPane.setVgap(5);
25     gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26     gridPane.add(tfAnnualInterestRate, 1, 0);
27     gridPane.add(new Label("Number of Years:"), 0, 1);
28     gridPane.add(tfNumberOfYears, 1, 1);
29     gridPane.add(new Label("Loan Amount:"), 0, 2);
30     gridPane.add(tfLoanAmount, 1, 2);
31     gridPane.add(new Label("Monthly Payment:"), 0, 3);
32     gridPane.add(tfMonthlyPayment, 1, 3);
33     gridPane.add(new Label("Total Payment:"), 0, 4);
34     gridPane.add(tfTotalPayment, 1, 4);
35     gridPane.add(btCalculate, 1, 5);
```



Annual Interest Rate: 4.5

Number of Years: 4

Loan Amount: 5000

Monthly Payment: $114.02

Total Payment: $5472.84

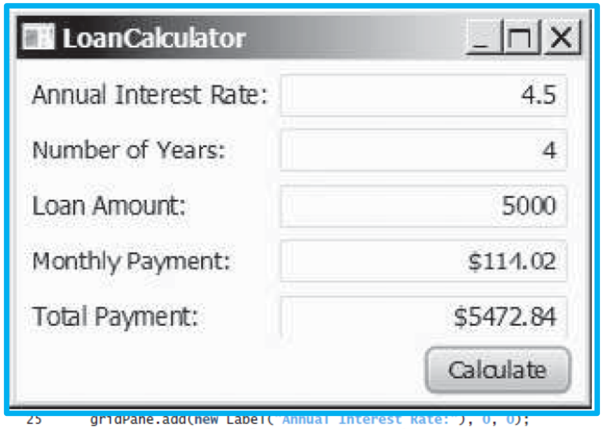
```
72     tfTotalPayment.setText(String.format("$%.2f",
73       loan.getTotalPayment()));
74   }
75 }
```

Line 20 begins our code, with an override of `start`

Lines 22 – 35 set up a 5-x-2 `GridPane`, with `Labels` (left side) and `TextFields` (right side) for the first 4 rows

The 5th row holds only the “Calculate” `Button`, right-justified in the right column (nothing in the left column)

§15.7: Case Study: Loan Calculator



Annual Interest Rate: 4.5

Number of Years: 4

Loan Amount: 5000

Monthly Payment: $114.02

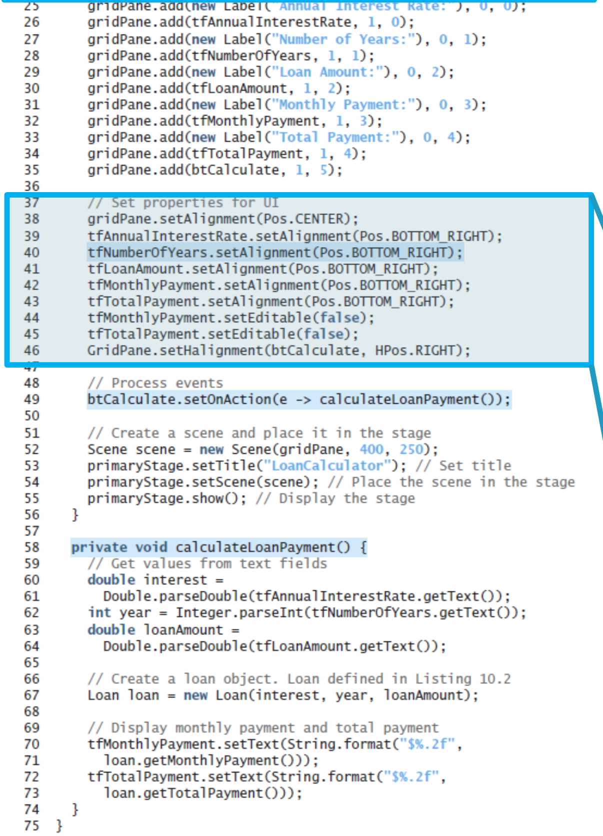
Total Payment: $5472.84

Line 38 centers the `GridPane` in its `Scene`

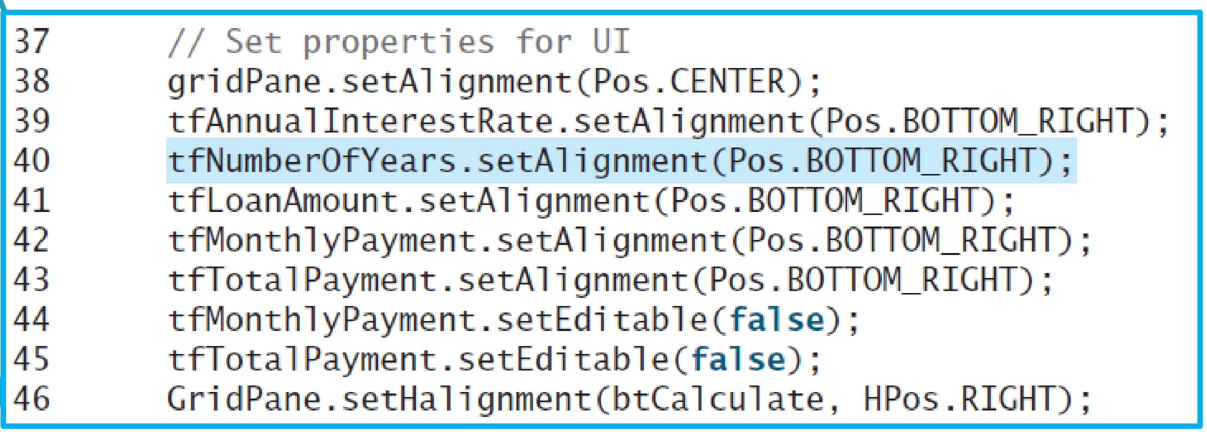
Lines 39 – 43 cause the text in the `TextFields` to be right-justified within the fields

Lines 44 – 45 keep the user from being able to edit (type inside of) those two `TextFields` (we're going to calculate their values and fill them in via code; the user shouldn't be able to enter anything in them)

Line 46 right-aligns the "Calculate" `Button` in its grid cell

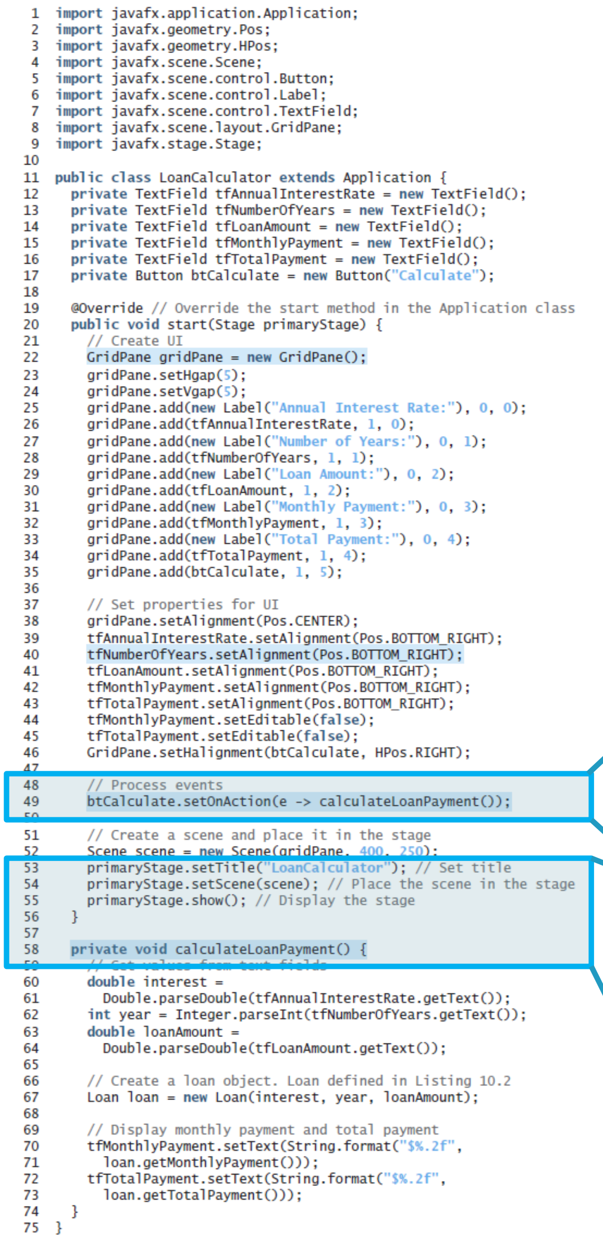


```
25      gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26      gridPane.add(tfAnnualInterestRate, 1, 0);
27      gridPane.add(new Label("Number of Years:"), 0, 1);
28      gridPane.add(tfNumberOfYears, 1, 1);
29      gridPane.add(new Label("Loan Amount:"), 0, 2);
30      gridPane.add(tfLoanAmount, 1, 2);
31      gridPane.add(new Label("Monthly Payment:"), 0, 3);
32      gridPane.add(tfMonthlyPayment, 1, 3);
33      gridPane.add(new Label("Total Payment:"), 0, 4);
34      gridPane.add(tfTotalPayment, 1, 4);
35      gridPane.add(btCalculate, 1, 5);
36
37      // Set properties for UI
38      gridPane.setAlignment(Pos.CENTER);
39      tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40      tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41      tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42      tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43      tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44      tfMonthlyPayment.setEditable(false);
45      tfTotalPayment.setEditable(false);
46      GridPane.setHalignment(btCalculate, HPos.RIGHT);
47
48      // Process events
49      btCalculate.setOnAction(e -> calculateLoanPayment());
50
51      // Create a scene and place it in the stage
52      Scene scene = new Scene(gridPane, 400, 250);
53      primaryStage.setTitle("LoanCalculator"); // Set title
54      primaryStage.setScene(scene); // Place the scene in the stage
55      primaryStage.show(); // Display the stage
56    }
57
58    private void calculateLoanPayment() {
59      // Get values from text fields
60      double interest =
61        Double.parseDouble(tfAnnualInterestRate.getText());
62      int year = Integer.parseInt(tfNumberOfYears.getText());
63      double loanAmount =
64        Double.parseDouble(tfLoanAmount.getText());
65
66      // Create a loan object. Loan defined in Listing 10.2
67      Loan loan = new Loan(interest, year, loanAmount);
68
69      // Display monthly payment and total payment
70      tfMonthlyPayment.setText(String.format("$%.2f",
71        loan.getMonthlyPayment()));
72      tfTotalPayment.setText(String.format("$%.2f",
73        loan.getTotalPayment()));
74    }
75  }
```



```
37      // Set properties for UI
38      gridPane.setAlignment(Pos.CENTER);
39      tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40      tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41      tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42      tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43      tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44      tfMonthlyPayment.setEditable(false);
45      tfTotalPayment.setEditable(false);
46      GridPane.setHalignment(btCalculate, HPos.RIGHT);
```

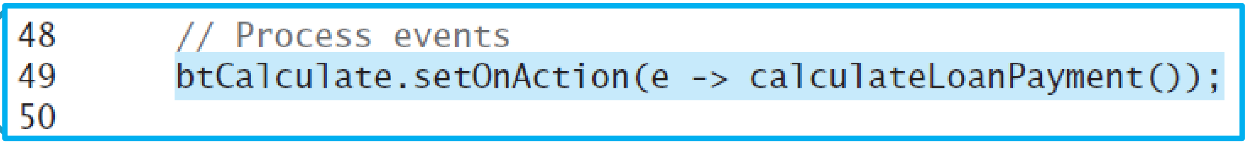
§15.7: Case Study: Loan Calculator



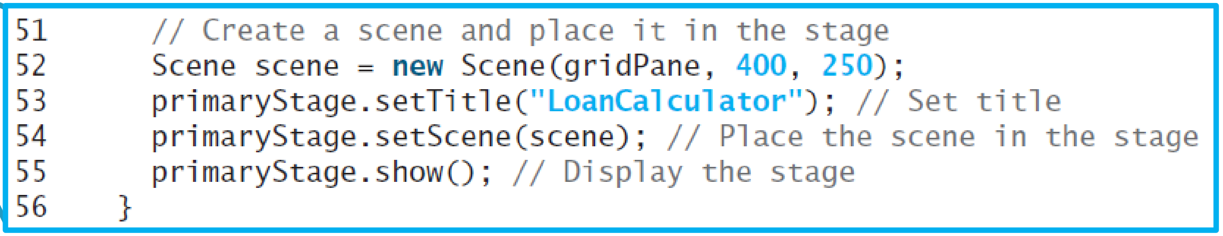
```
1  import javafx.application.Application;
2  import javafx.geometry.Pos;
3  import javafx.geometry.HPos;
4  import javafx.scene.Scene;
5  import javafx.scene.control.Button;
6  import javafx.scene.control.Label;
7  import javafx.scene.control.TextField;
8  import javafx.scene.layout.GridPane;
9  import javafx.stage.Stage;
10
11 public class LoanCalculator extends Application {
12   private TextField tfAnnualInterestRate = new TextField();
13   private TextField tfNumberOfYears = new TextField();
14   private TextField tfLoanAmount = new TextField();
15   private TextField tfMonthlyPayment = new TextField();
16   private TextField tfTotalPayment = new TextField();
17   private Button btCalculate = new Button("Calculate");
18
19   @Override // Override the start method in the Application class
20   public void start(Stage primaryStage) {
21     // Create UI
22     GridPane gridPane = new GridPane();
23     gridPane.setHgap(5);
24     gridPane.setVgap(5);
25     gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26     gridPane.add(tfAnnualInterestRate, 1, 0);
27     gridPane.add(new Label("Number of Years:"), 0, 1);
28     gridPane.add(tfNumberOfYears, 1, 1);
29     gridPane.add(new Label("Loan Amount:"), 0, 2);
30     gridPane.add(tfLoanAmount, 1, 2);
31     gridPane.add(new Label("Monthly Payment:"), 0, 3);
32     gridPane.add(tfMonthlyPayment, 1, 3);
33     gridPane.add(new Label("Total Payment:"), 0, 4);
34     gridPane.add(tfTotalPayment, 1, 4);
35     gridPane.add(btCalculate, 1, 5);
36
37     // Set properties for UI
38     gridPane.setAlignment(Pos.CENTER);
39     tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40     tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41     tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42     tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43     tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44     tfMonthlyPayment.setEditable(false);
45     tfTotalPayment.setEditable(false);
46     GridPane.setHalignment(btCalculate, HPos.RIGHT);
47
48     // Process events
49     btCalculate.setOnAction(e -> calculateLoanPayment());
50
51     // Create a scene and place it in the stage
52     Scene scene = new Scene(gridPane, 400, 250);
53     primaryStage.setTitle("LoanCalculator"); // Set title
54     primaryStage.setScene(scene); // Place the scene in the stage
55     primaryStage.show(); // Display the stage
56   }
57
58   private void calculateLoanPayment() {
59     // Get values from text fields
60     double interest =
61       Double.parseDouble(tfAnnualInterestRate.getText());
62     int year = Integer.parseInt(tfNumberOfYears.getText());
63     double loanAmount =
64       Double.parseDouble(tfLoanAmount.getText());
65
66     // Create a loan object. Loan defined in Listing 10.2
67     Loan loan = new Loan(interest, year, loanAmount);
68
69     // Display monthly payment and total payment
70     tfMonthlyPayment.setText(String.format("$%.2f",
71       loan.getMonthlyPayment()));
72     tfTotalPayment.setText(String.format("$%.2f",
73       loan.getTotalPayment()));
74   }
75 }
```

Line 49 uses a Lambda Expression to register the "Calculate" `Button` to its `handle` method (located in the implicit anonymous inner class), which contains a single line of code – a call to the `calculateLoanPayment` method

Lines 51 – 56 form the typical end of the `start` method: The container for our UI elements (the `GridPane`) is added to a new `Scene` (l. 52), which is added to the `Stage` (l. 54).
The `Stage` is given a title (l. 53), and made visible (l. 55)



```
48      // Process events
49      btCalculate.setOnAction(e -> calculateLoanPayment());
50
```



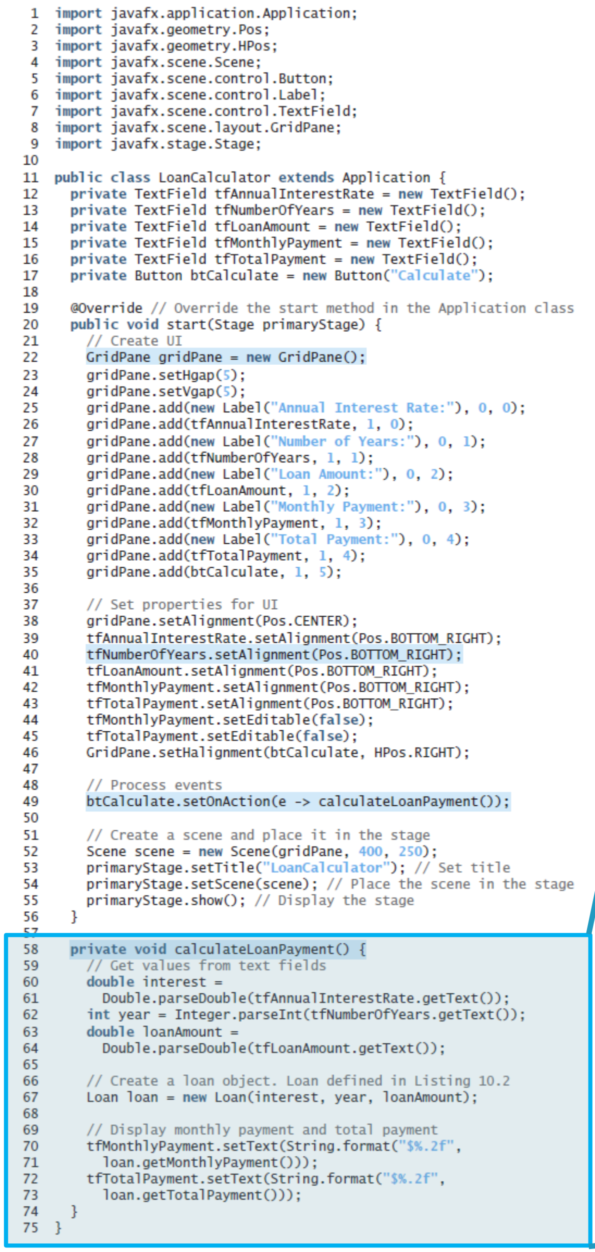
```
51      // Create a scene and place it in the stage
52      Scene scene = new Scene(gridPane, 400, 250);
53      primaryStage.setTitle("LoanCalculator"); // Set title
54      primaryStage.setScene(scene); // Place the scene in the stage
55      primaryStage.show(); // Display the stage
56    }
```

§15.7: Case Study: Loan Calculator

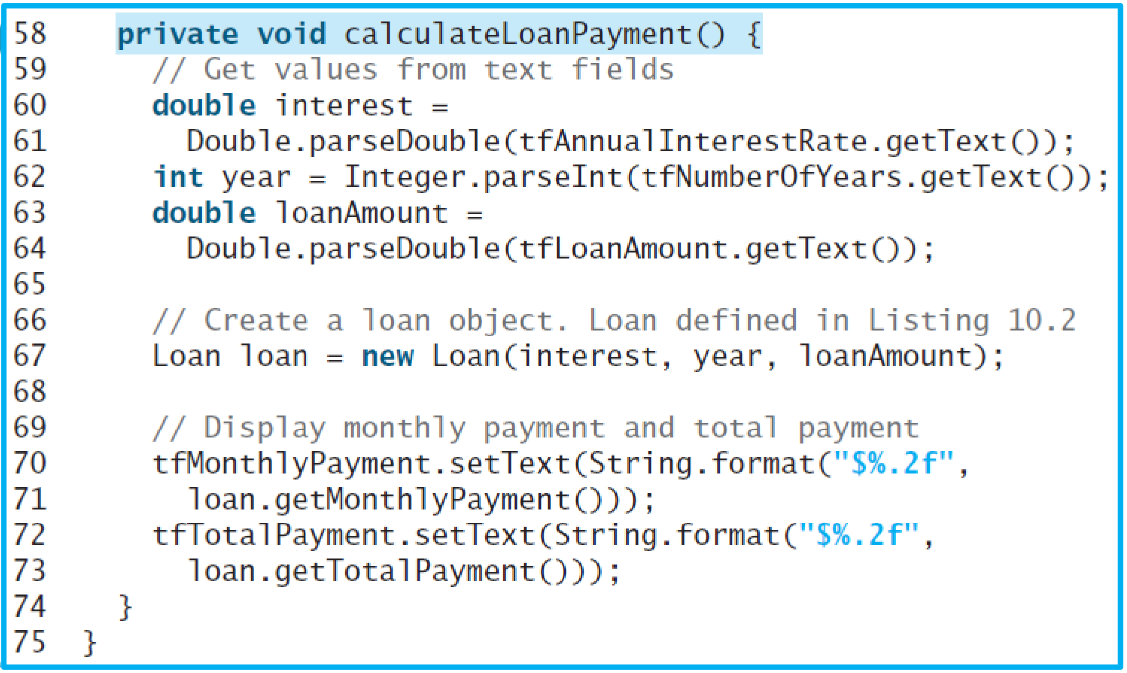
Line 49 registers a call to `calculateLoanPayment` as the handler for the “Calculate” `Button`’s `ActionEvent`.

This is that code.

In order to do the calculations, we need the values to be in numeric variables, but the `TextField` boxes all hold `Strings`, so we have to parse them into numeric variables first (ll. 60-64)

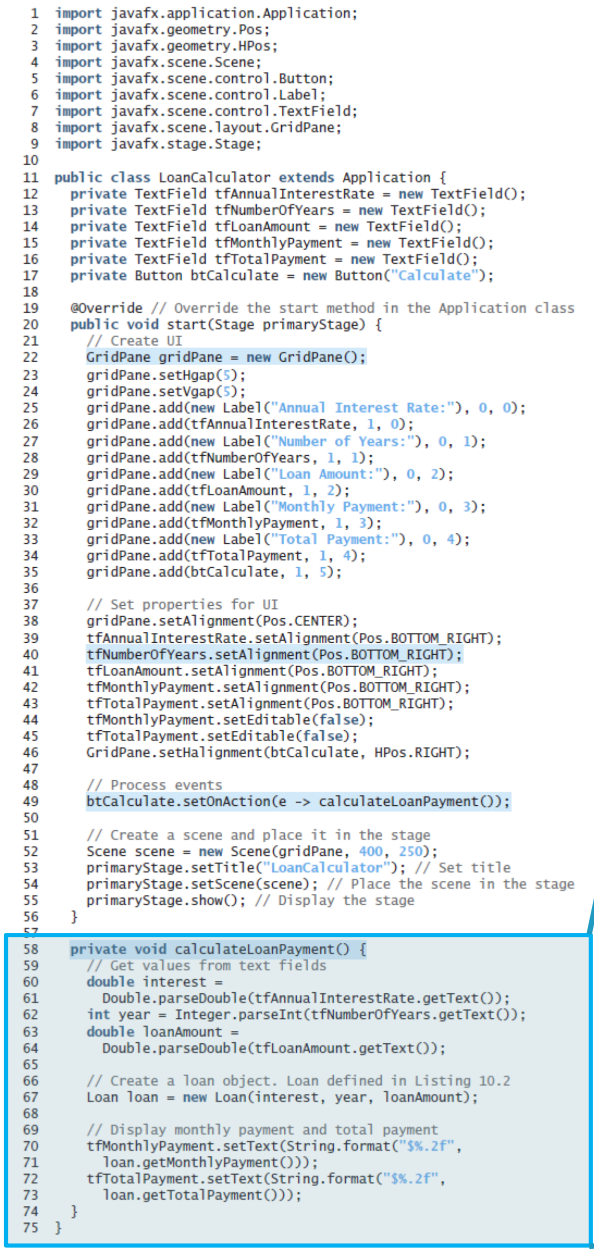


```
 1  import javafx.application.Application;
 2  import javafx.geometry.Pos;
 3  import javafx.geometry.HPos;
 4  import javafx.scene.Scene;
 5  import javafx.scene.control.Button;
 6  import javafx.scene.control.Label;
 7  import javafx.scene.control.TextField;
 8  import javafx.scene.layout.GridPane;
 9  import javafx.stage.Stage;
10
11  public class LoanCalculator extends Application {
12    private TextField tfAnnualInterestRate = new TextField();
13    private TextField tfNumberOfYears = new TextField();
14    private TextField tfLoanAmount = new TextField();
15    private TextField tfMonthlyPayment = new TextField();
16    private TextField tfTotalPayment = new TextField();
17    private Button btCalculate = new Button("Calculate");
18
19    @Override // Override the start method in the Application class
20    public void start(Stage primaryStage) {
21      // Create UI
22      GridPane gridPane = new GridPane();
23      gridPane.setHgap(5);
24      gridPane.setVgap(5);
25      gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26      gridPane.add(tfAnnualInterestRate, 1, 0);
27      gridPane.add(new Label("Number of Years:"), 0, 1);
28      gridPane.add(tfNumberOfYears, 1, 1);
29      gridPane.add(new Label("Loan Amount:"), 0, 2);
30      gridPane.add(tfLoanAmount, 1, 2);
31      gridPane.add(new Label("Monthly Payment:"), 0, 3);
32      gridPane.add(tfMonthlyPayment, 1, 3);
33      gridPane.add(new Label("Total Payment:"), 0, 4);
34      gridPane.add(tfTotalPayment, 1, 4);
35      gridPane.add(btCalculate, 1, 5);
36
37      // Set properties for UI
38      gridPane.setAlignment(Pos.CENTER);
39      tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40      tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41      tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42      tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43      tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44      tfMonthlyPayment.setEditable(false);
45      tfTotalPayment.setEditable(false);
46      GridPane.setHalignment(btCalculate, HPos.RIGHT);
47
48      // Process events
49      btCalculate.setOnAction(e -> calculateLoanPayment());
50
51      // Create a scene and place it in the stage
52      Scene scene = new Scene(gridPane, 400, 250);
53      primaryStage.setTitle("LoanCalculator"); // Set title
54      primaryStage.setScene(scene); // Place the scene in the stage
55      primaryStage.show(); // Display the stage
56    }
57
58    private void calculateLoanPayment() {
59      // Get values from text fields
60      double interest =
61        Double.parseDouble(tfAnnualInterestRate.getText());
62      int year = Integer.parseInt(tfNumberOfYears.getText());
63      double loanAmount =
64        Double.parseDouble(tfLoanAmount.getText());
65
66      // Create a loan object. Loan defined in Listing 10.2
67      Loan loan = new Loan(interest, year, loanAmount);
68
69      // Display monthly payment and total payment
70      tfMonthlyPayment.setText(String.format("$%.2f",
71        loan.getMonthlyPayment()));
72      tfTotalPayment.setText(String.format("$%.2f",
73        loan.getTotalPayment()));
74    }
75  }
```



```
58    private void calculateLoanPayment() {
59      // Get values from text fields
60      double interest =
61        Double.parseDouble(tfAnnualInterestRate.getText());
62      int year = Integer.parseInt(tfNumberOfYears.getText());
63      double loanAmount =
64        Double.parseDouble(tfLoanAmount.getText());
65
66      // Create a loan object. Loan defined in Listing 10.2
67      Loan loan = new Loan(interest, year, loanAmount);
68
69      // Display monthly payment and total payment
70      tfMonthlyPayment.setText(String.format("$%.2f",
71        loan.getMonthlyPayment()));
72      tfTotalPayment.setText(String.format("$%.2f",
73        loan.getTotalPayment()));
74    }
75  }
```

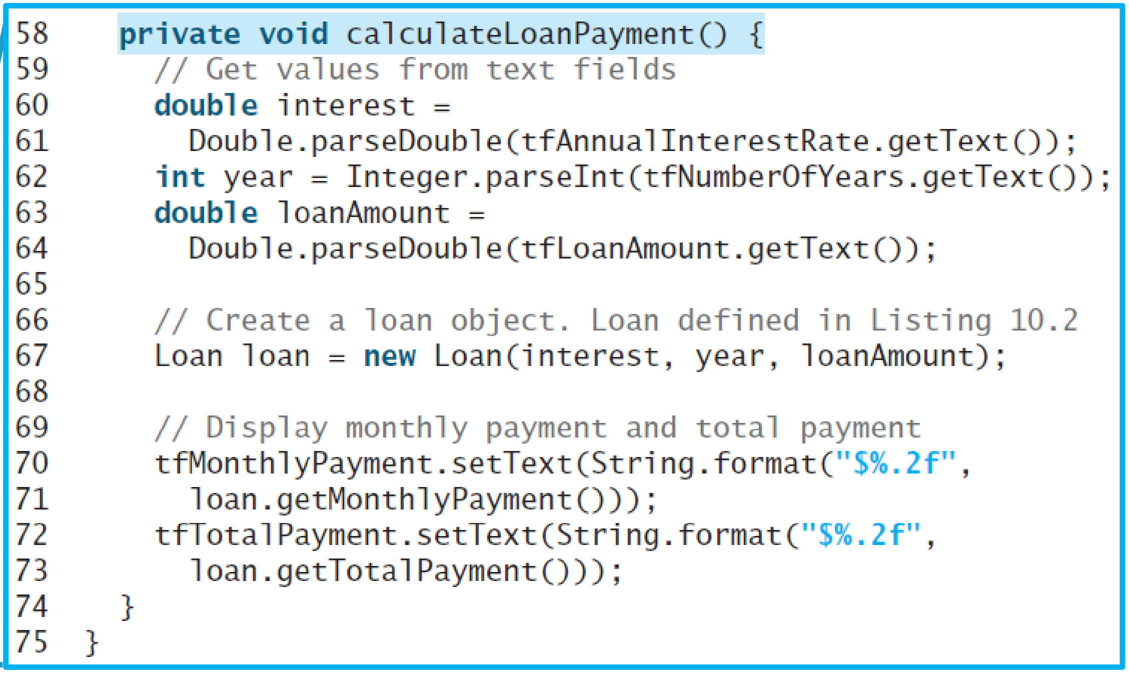
§15.7: Case Study: Loan Calculator



```
 1 import javafx.application.Application;
 2 import javafx.geometry.Pos;
 3 import javafx.geometry.HPos;
 4 import javafx.scene.Scene;
 5 import javafx.scene.control.Button;
 6 import javafx.scene.control.Label;
 7 import javafx.scene.control.TextField;
 8 import javafx.scene.layout.GridPane;
 9 import javafx.stage.Stage;
10
11 public class LoanCalculator extends Application {
12   private TextField tfAnnualInterestRate = new TextField();
13   private TextField tfNumberOfYears = new TextField();
14   private TextField tfLoanAmount = new TextField();
15   private TextField tfMonthlyPayment = new TextField();
16   private TextField tfTotalPayment = new TextField();
17   private Button btCalculate = new Button("Calculate");
18
19   @Override // Override the start method in the Application class
20   public void start(Stage primaryStage) {
21     // Create UI
22     GridPane gridPane = new GridPane();
23     gridPane.setHgap(5);
24     gridPane.setVgap(5);
25     gridPane.add(new Label("Annual Interest Rate:"), 0, 0);
26     gridPane.add(tfAnnualInterestRate, 1, 0);
27     gridPane.add(new Label("Number of Years:"), 0, 1);
28     gridPane.add(tfNumberOfYears, 1, 1);
29     gridPane.add(new Label("Loan Amount:"), 0, 2);
30     gridPane.add(tfLoanAmount, 1, 2);
31     gridPane.add(new Label("Monthly Payment:"), 0, 3);
32     gridPane.add(tfMonthlyPayment, 1, 3);
33     gridPane.add(new Label("Total Payment:"), 0, 4);
34     gridPane.add(tfTotalPayment, 1, 4);
35     gridPane.add(btCalculate, 1, 5);
36
37     // Set properties for UI
38     gridPane.setAlignment(Pos.CENTER);
39     tfAnnualInterestRate.setAlignment(Pos.BOTTOM_RIGHT);
40     tfNumberOfYears.setAlignment(Pos.BOTTOM_RIGHT);
41     tfLoanAmount.setAlignment(Pos.BOTTOM_RIGHT);
42     tfMonthlyPayment.setAlignment(Pos.BOTTOM_RIGHT);
43     tfTotalPayment.setAlignment(Pos.BOTTOM_RIGHT);
44     tfMonthlyPayment.setEditable(false);
45     tfTotalPayment.setEditable(false);
46     GridPane.setHalignment(btCalculate, HPos.RIGHT);
47
48     // Process events
49     btCalculate.setOnAction(e -> calculateLoanPayment());
50
51     // Create a scene and place it in the stage
52     Scene scene = new Scene(gridPane, 400, 250);
53     primaryStage.setTitle("LoanCalculator"); // Set title
54     primaryStage.setScene(scene); // Place the scene in the stage
55     primaryStage.show(); // Display the stage
56   }
57
58   private void calculateLoanPayment() {
59     // Get values from text fields
60     double interest =
61       Double.parseDouble(tfAnnualInterestRate.getText());
62     int year = Integer.parseInt(tfNumberOfYears.getText());
63     double loanAmount =
64       Double.parseDouble(tfLoanAmount.getText());
65
66     // Create a loan object. Loan defined in Listing 10.2
67     Loan loan = new Loan(interest, year, loanAmount);
68
69     // Display monthly payment and total payment
70     tfMonthlyPayment.setText(String.format("$%.2f",
71       loan.getMonthlyPayment()));
72     tfTotalPayment.setText(String.format("$%.2f",
73       loan.getTotalPayment()));
74   }
75 }
```

Once we have the numeric versions of the `Strings` in the three `TextFields`, we can use those to create a `Loan` object (see Listing 10.2, pp. 368 – 369) – l. 67

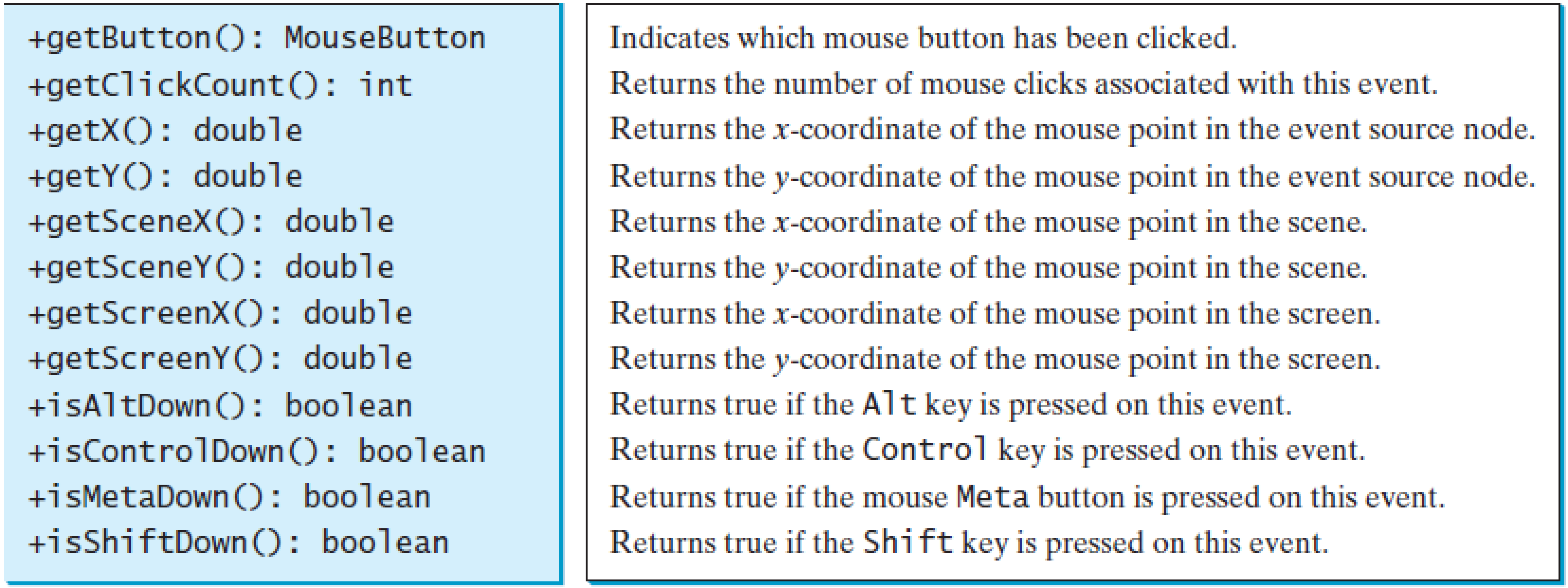
Once instantiated, the `Loan` class provides the methods `getMonthlyPayment()` and `getTotalPayment()` to give us the values to display in the last two `TextFields` (ll. 70 – 73).



```
58   private void calculateLoanPayment() {
59     // Get values from text fields
60     double interest =
61       Double.parseDouble(tfAnnualInterestRate.getText());
62     int year = Integer.parseInt(tfNumberOfYears.getText());
63     double loanAmount =
64       Double.parseDouble(tfLoanAmount.getText());
65
66     // Create a loan object. Loan defined in Listing 10.2
67     Loan loan = new Loan(interest, year, loanAmount);
68
69     // Display monthly payment and total payment
70     tfMonthlyPayment.setText(String.format("$%.2f",
71       loan.getMonthlyPayment()));
72     tfTotalPayment.setText(String.format("$%.2f",
73       loan.getTotalPayment()));
74   }
75 }
```

§15.8: Mouse Events

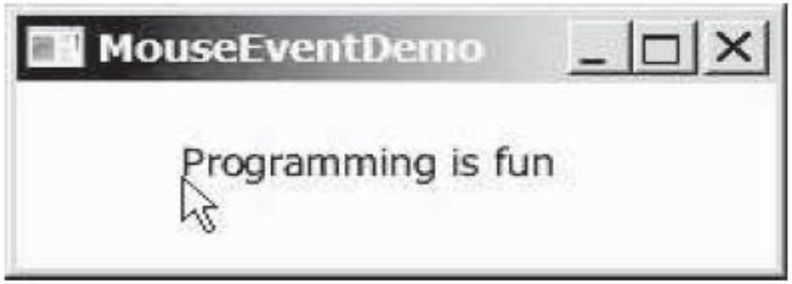
- Mouse events are fired whenever the mouse…
  - …button goes down (MousePressed)
  - …button comes back up (MouseReleased)
  - …button is clicked (a down/up cycle – MouseClicked)
  - …first enters an element (MouseEntered)
  - …leaves an element (MouseExited)
  - …moves while over an element (MouseMoved)
  - …is dragged (moved with the mouse button held down – MouseDragged)
- All mouse events have some common methods we can use to tell more about the state / location of the mouse and the keyboard's Alt / Control / Shift keys:



+getButton(): MouseButton	Indicates which mouse button has been clicked.
+getClickCount(): int	Returns the number of mouse clicks associated with this event.
+getX(): double	Returns the *x*-coordinate of the mouse point in the event source node.
+getY(): double	Returns the *y*-coordinate of the mouse point in the event source node.
+getSceneX(): double	Returns the *x*-coordinate of the mouse point in the scene.
+getSceneY(): double	Returns the *y*-coordinate of the mouse point in the scene.
+getScreenX(): double	Returns the *x*-coordinate of the mouse point in the screen.
+getScreenY(): double	Returns the *y*-coordinate of the mouse point in the screen.
+isAltDown(): boolean	Returns true if the Alt key is pressed on this event.
+isControlDown(): boolean	Returns true if the Control key is pressed on this event.
+isMetaDown(): boolean	Returns true if the mouse Meta button is pressed on this event.
+isShiftDown(): boolean	Returns true if the Shift key is pressed on this event.

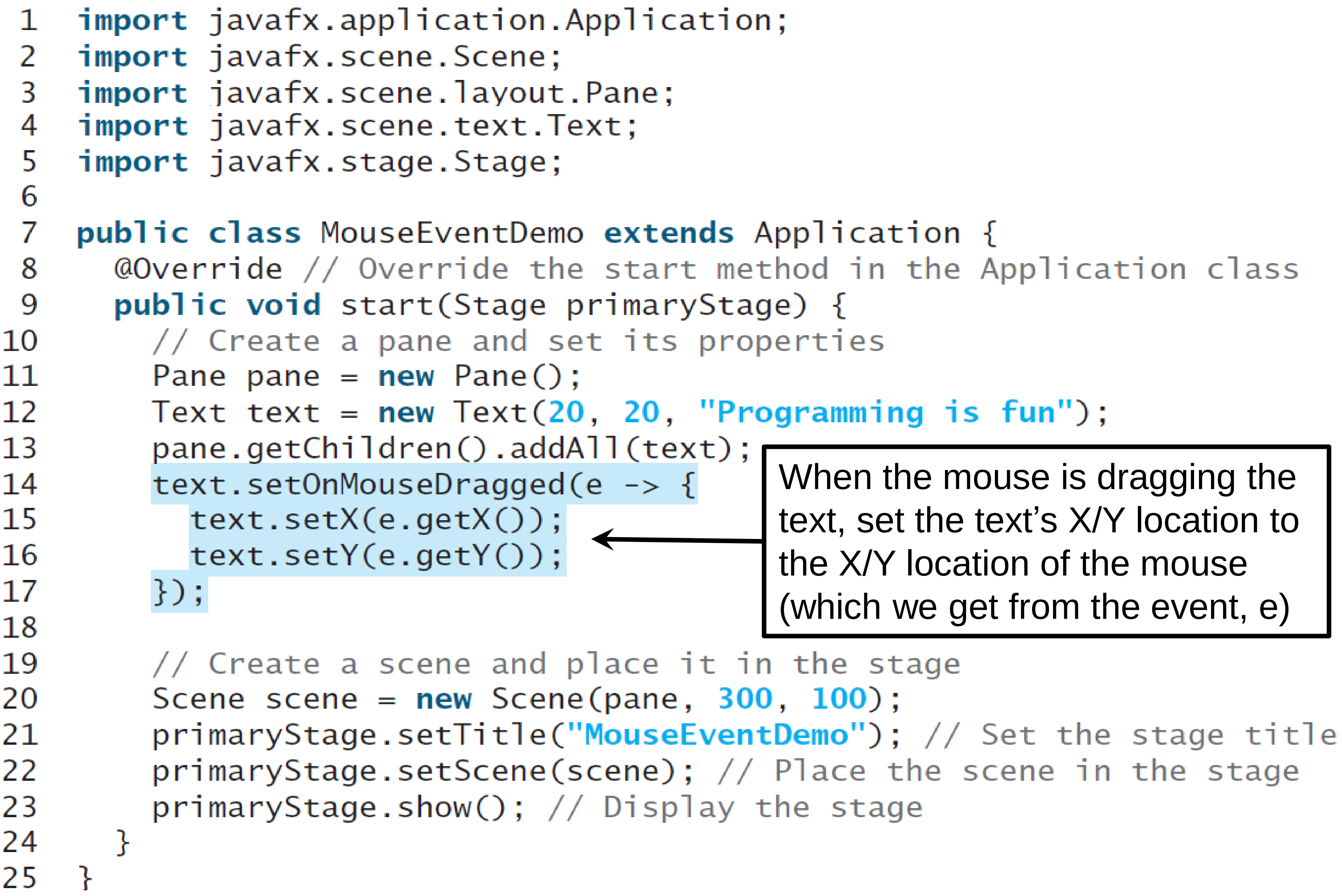
§15.8: Mouse Events

- JavaFX (currently) supports 3 buttons, left, middle, and right (some mice have more)
- We can tell *which* mouse button was pressed by using the `MouseEvent`'s `getButton()` method and comparing the result to: `MouseButton.PRIMARY`, `MouseButton.SECONDARY`, `MouseButton.MIDDLE`, or `MouseButton.NONE`
  - `NONE` is on the list simply because some mouse events (like `MouseMoved`) don't involve a button
- The book presents a sample program that lets the user drag (click, holding the primary (left?) mouse button down, and moving the mouse before releasing the button) a `Text` node around on the pane:



Programming is fun

§15.8: Mouse Events



```java
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.text.Text;
import javafx.stage.Stage;

public class MouseEventDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    Pane pane = new Pane();
    Text text = new Text(20, 20, "Programming is fun");
    pane.getChildren().addAll(text);
    text.setOnMouseDragged(e -> {
      text.setX(e.getX());
      text.setY(e.getY());
    });

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane, 300, 100);
    primaryStage.setTitle("MouseEventDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }
}
```

When the mouse is dragging the text, set the text's X/Y location to the X/Y location of the mouse (which we get from the event, e)

§15.9: Key Events

- The user can also interact with the GUI through the keyboard.
- We can handle events that are generated by the keyboard on a key-by-key basis, without having to wait for the user to press Enter, as we do with the Scanner
- *Key events* enable the use of the keys to control and perform actions or get input from the keyboard. The **KeyEvent** object describes the nature of the event (namely, that a key has been pressed, released, or typed) and the value of the key
- We can tell when keys go down (KEY_PRESSED), come back up (KEY_RELEASED)
- We can also tell if the SHIFT, ALT, CONTROL, or META (Mac) keys are down at the same time (isShiftDown(), isAltDown(), …)
- Not all keys generate a character we can display (some don’t generate a character *at all*!)

KeyCode Constants

Every key event has an associated code that is returned by the **getCode()** method in **KeyEvent**. The *key codes* are constants defined in **KeyCode**.

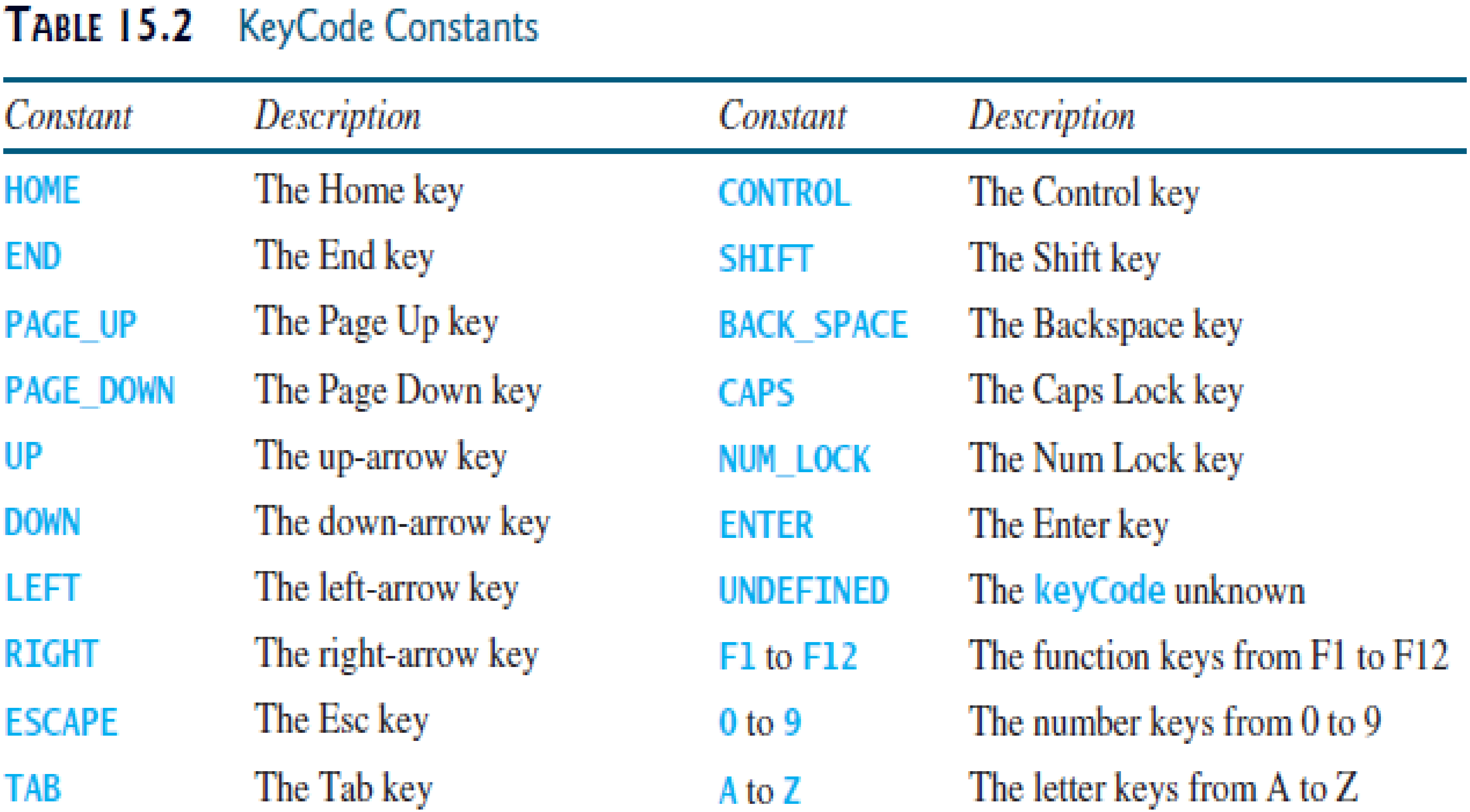
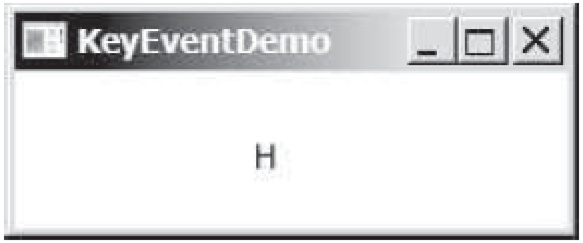


TABLE 15.2 KeyCode Constants

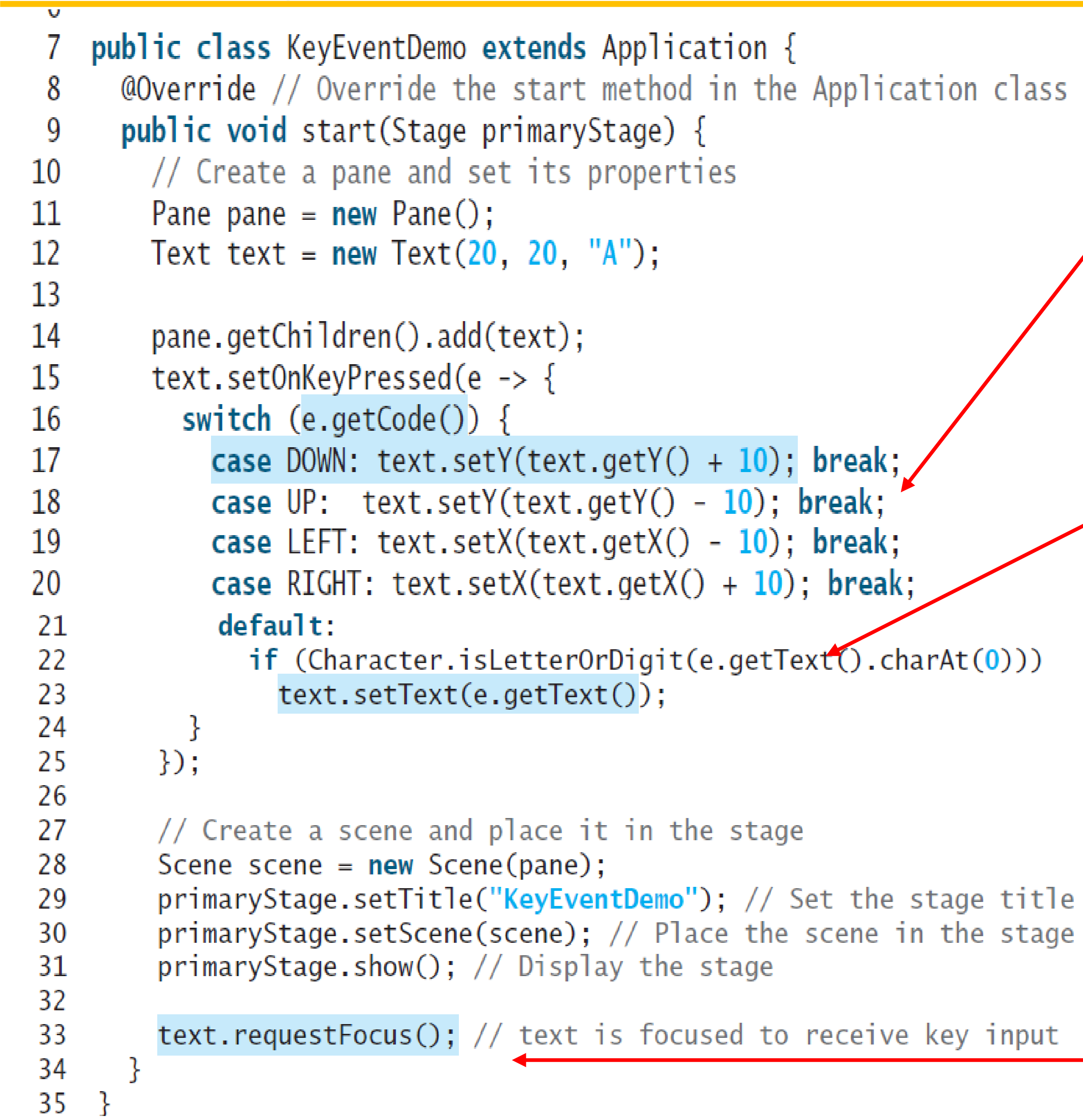
Constant	*Description*	*Constant*	*Description*
HOME	The Home key	CONTROL	The Control key
END	The End key	SHIFT	The Shift key
PAGE_UP	The Page Up key	BACK_SPACE	The Backspace key
PAGE_DOWN	The Page Down key	CAPS	The Caps Lock key
UP	The up-arrow key	NUM_LOCK	The Num Lock key
DOWN	The down-arrow key	ENTER	The Enter key
LEFT	The left-arrow key	UNDEFINED	The keyCode unknown
RIGHT	The right-arrow key	F1 to F12	The function keys from F1 to F12
ESCAPE	The Esc key	0 to 9	The number keys from 0 to 9
TAB	The Tab key	A to Z	The letter keys from A to Z

§15.9: Key Events - Demo

- The `KeyEventDemo` program (Listing 15.8, pp. 604 – 605, and next slide) starts with a window (Stage) containing "A"
- If the user types a letter or digit, it replaces the character displayed.
- If the user presses the `UP`, `DOWN`, `LEFT`, or `RIGHT` cursor-movement keys, the letter moves in the corresponding direction by 10 pixels.



§15.9: Key Events - Demo



```java
public class KeyEventDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    Pane pane = new Pane();
    Text text = new Text(20, 20, "A");

    pane.getChildren().add(text);
    text.setOnKeyPressed(e -> {
      switch (e.getCode()) {
        case DOWN: text.setY(text.getY() + 10); break;
        case UP:  text.setY(text.getY() - 10); break;
        case LEFT: text.setX(text.getX() - 10); break;
        case RIGHT: text.setX(text.getX() + 10); break;
        default:
          if (Character.isLetterOrDigit(e.getText().charAt(0)))
            text.setText(e.getText());
      }
    });

    // Create a scene and place it in the stage
    Scene scene = new Scene(pane);
    primaryStage.setTitle("KeyEventDemo"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage

    text.requestFocus(); // text is focused to receive key input
  }
}
```

When our `Text` node receives a `KeyPressed` event, if it's `UP`, `DOWN`, `LEFT`, or `RIGHT`, move the text 10 pixels in the proper direction.

If it's none of those four, but it IS a letter or digit, then change the content of the `Text` node to whatever the key was.

How do we know our `Text` node (as opposed to some *other* node, if we had more on the pane) will receive the key press?

We give it the ***focus***
(see p. 605)