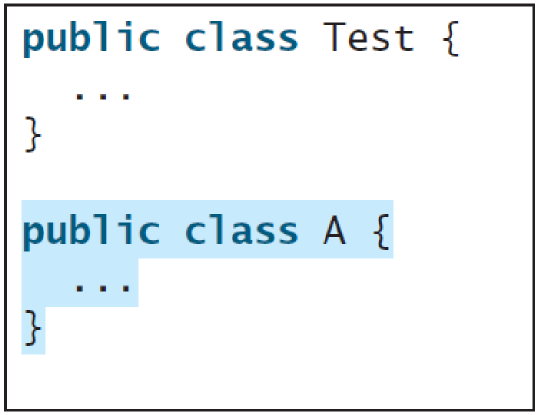
Chapter 15 Event-Driven Programming and Animations

Inner class handlers
Anonymous Inner class handlers
lambda expressions

Inner Classes

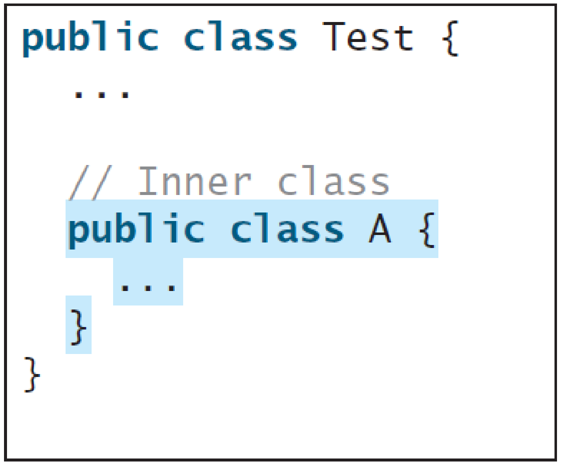
Inner class: A class is a member of another class.

Advantages: In some applications, you can use an inner class to make programs simple.



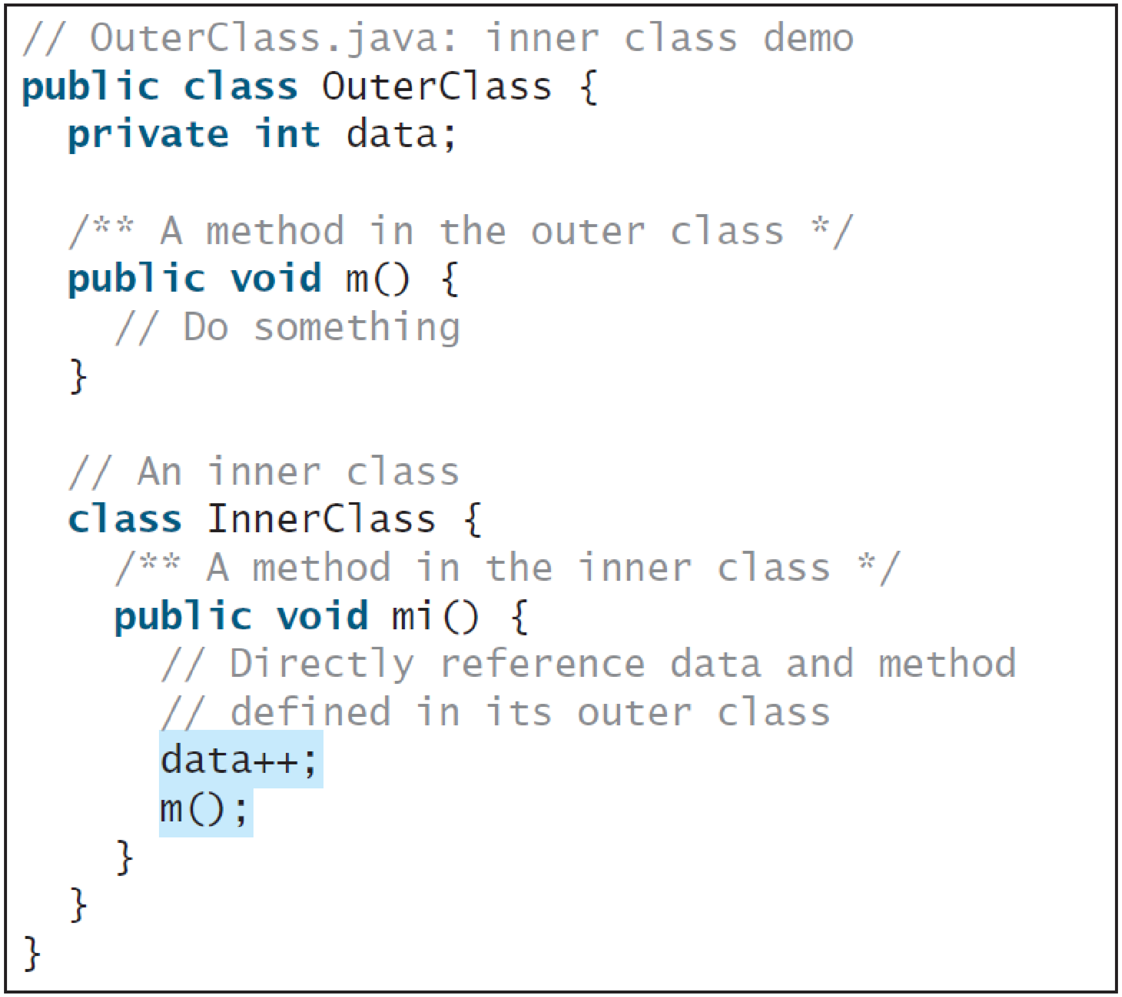
```
public class Test {
  ...
}

public class A {
  ...
}
```

(a)



```
public class Test {
  ...

  // Inner class
  public class A {
    ...
  }
}
```

(b)



```
// 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();
    }
  }
}
```

(c)

Inner Classes (cont.)

- **Inner classes has the following features:**

1. An inner class is compiled into a class named: *OuterClassName*$*InnerClassName*.class.

For example, the inner class A in outer class Test is compiled into *Test$A*.class .

2. An inner class can reference the data and methods defined in the outer class in which it nests, so you do not need to pass the reference of the outer class to the constructor of the inner class.

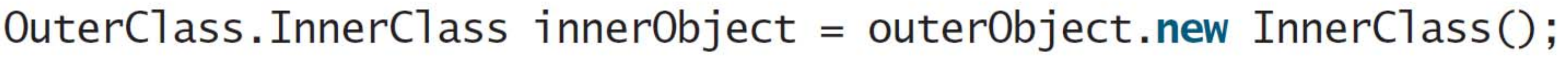
Inner Classes (cont.)

3. An inner class can be declared public, protected, or private subject to the same visibility rules applied to a member of the class.

4. An inner class can be declared 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.

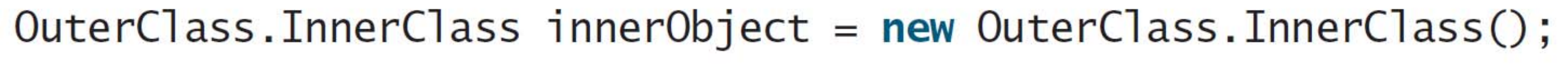
5. If the inner class is public, you can create an object of the inner class from another class.

If the inner class is nonstatic



```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

If the inner class is static



```
OuterClass.InnerClass innerObject = new OuterClass.InnerClass();
```

Inner Classes (cont.)

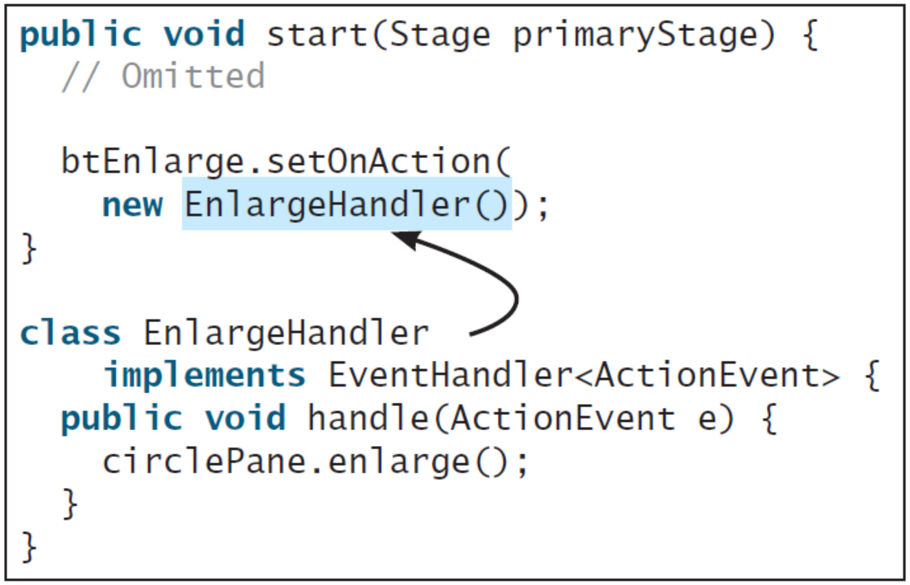
- A simple use of inner classes is to combine dependent classes into a primary class.
- This reduces the number of source files.
- It also makes class files easy to organize since they are all named with the primary class as the prefix.
- For example, rather than creating the two source files Test.java and A.java, you can merge class A into class Test and create just one source file, Test.java. The resulting class files are Test.class and Test$A.class.

Inner Class Handlers

An event handler class is designed specifically to create a handler object for a GUI component (e.g., a button). It will not be shared by other applications. So, it is appropriate to define the Event Handler class inside the Application class as an inner class.

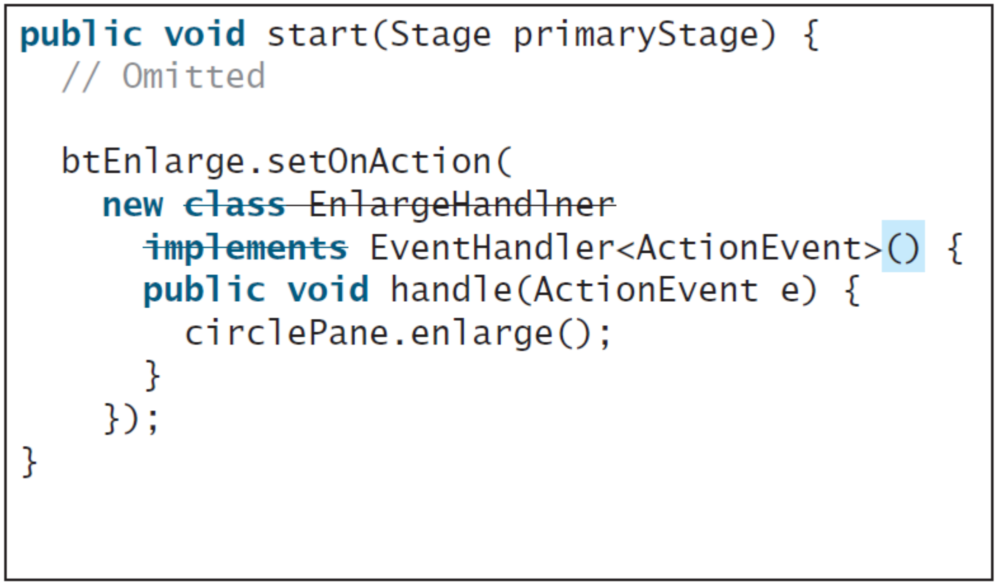
Anonymous Inner Classes

- 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.



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

  btEnlarge.setOnAction(
    new EnlargeHandler());
}

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

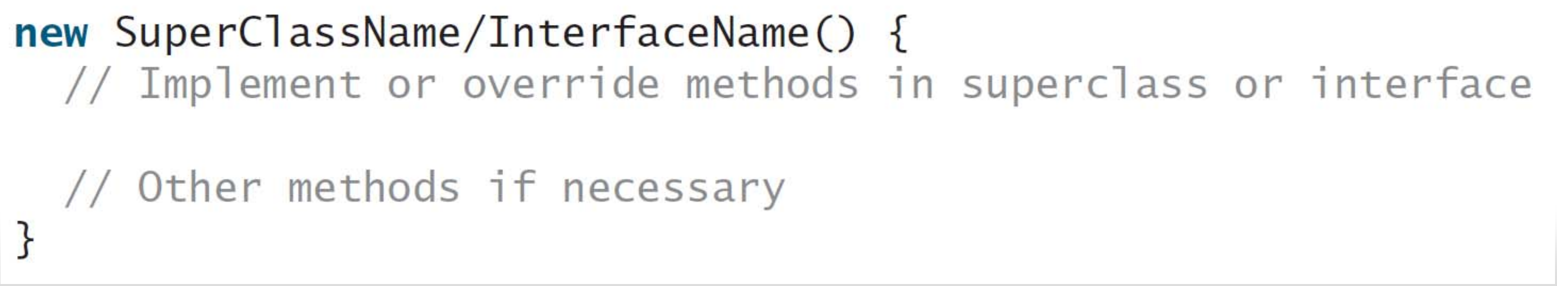
(a) Inner class `EnlargeListener`



```
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

- The syntax for an anonymous inner class is:



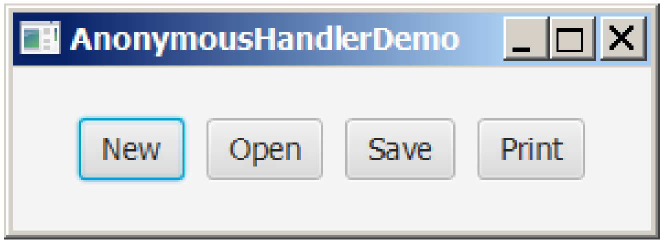
```
new SuperClassName/InterfaceName() {
  // Implement or override methods in superclass or interface

  // Other methods if necessary
}
```

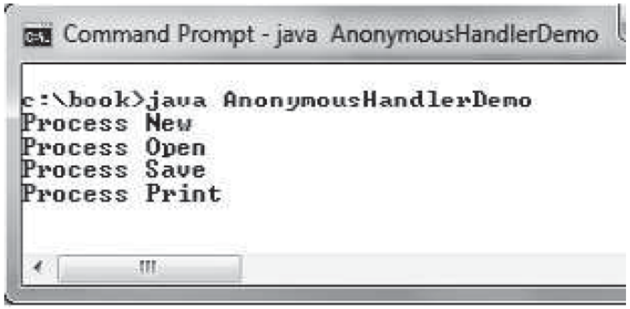
Anonymous Inner Classes

- An anonymous inner class must always extend a superclass or implement an interface, but it cannot have an explicit **extends** or **implements** clause.
- An anonymous inner class must implement all the abstract methods in the superclass or in the interface.
- 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**().
- 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, these two classes are compiled into **Test$1.class** and **Test$2.class**.

Anonymous Inner Classes Example

```
public class AnonymousHandlerDemo extends Application {
  @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 btNew = new Button("New");
    Button btOpen = new Button("Open");
    Button btSave = new Button("Save");
    Button btPrint = new Button("Print");
    hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);

    // 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");
      }
    });
```





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

Anonymous Inner Classes Example

- Without using anonymous inner classes, we would have to create four separate classes.
- An anonymous handler works the same way as that of an inner class handler. The program is condensed using an anonymous inner class.
- The anonymous inner classes in this example are compiled into:
- **AnonymousHandlerDemo$1.class**, **AnonymousHandlerDemo$2.class**, **AnonymousHandlerDemo$3.class**, and **AnonymousHandlerDemo$4.class**.

Simplifying Event Handing Using Lambda Expressions

Lambda expression is a new feature in Java 8. Lambda expressions can be viewed as an anonymous class with a concise syntax. For example, the following code in (a) can be greatly simplified using a lambda expression in (b) in three lines.

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

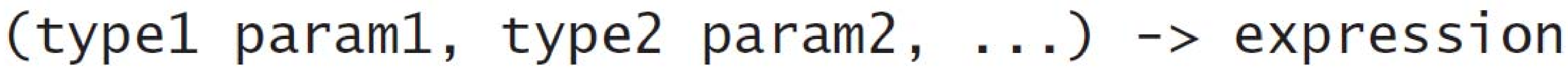
(a) Anonymous inner class event handler

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

(b) Lambda expression event handler

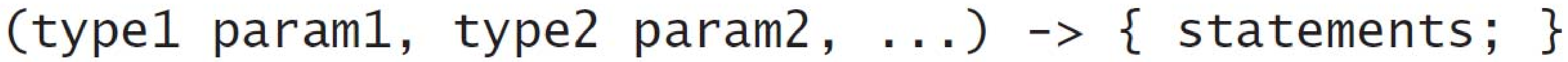
Basic Syntax for a Lambda Expression

The basic syntax for a lambda expression is either



```
(type1 param1, type2 param2, ...) -> expression
```

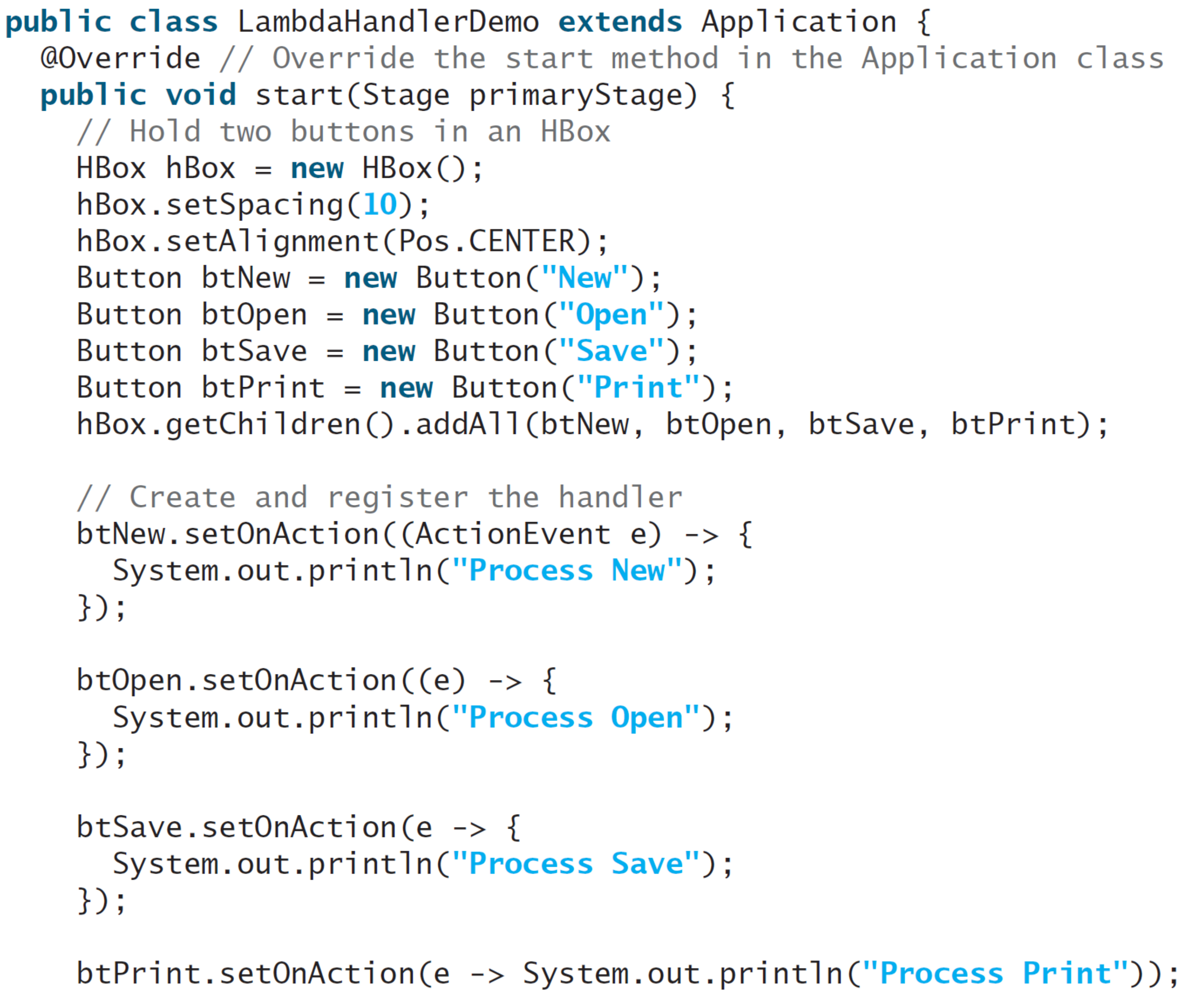
or



```
(type1 param1, type2 param2, ...) -> { statements; }
```

The data type for a parameter may be explicitly declared or implicitly inferred by the compiler. The parentheses can be omitted if there is only one parameter without an explicit data type.

Lambda Expressions Example



```
public class LambdaHandlerDemo extends Application {
  @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 btNew = new Button("New");
    Button btOpen = new Button("Open");
    Button btSave = new Button("Save");
    Button btPrint = new Button("Print");
    hBox.getChildren().addAll(btNew, btOpen, btSave, btPrint);

    // 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"));
```

Lambda Expressions

- The compiler treats a lambda expression as if it is an object created from an anonymous inner class.
- In this case, the compiler understands that the object must be an instance of **EventHandler**<**ActionEvent**>.
- Since the **EventHandler** interface defines the handle method with a parameter of the **ActionEvent** type, the compiler automatically recognizes that **e** is a parameter of the **ActionEvent** type, and the statements are for the body of the **handle** method.

Lambda Expressions

- The **EventHandler** interface contains just one method. The statements in the lambda expression are all for that method.
- If it contains multiple methods, the compiler will not be able to compile the lambda expression.
- So, for the compiler to understand lambda expressions, the interface must contain exactly one **abstract** method.
- Such an interface is known as a *functional interface* or a *Single Abstract Method* (SAM) interface.

Mouse Events

- A **MouseEvent** is fired whenever a mouse button is pressed, released, clicked, moved, or dragged on a node or a scene.
- The **MouseEvent** object captures the event, such as the number of clicks associated with it, the location (the x- and y-coordinates) of the mouse, which mouse button was pressed …

Mouse Events

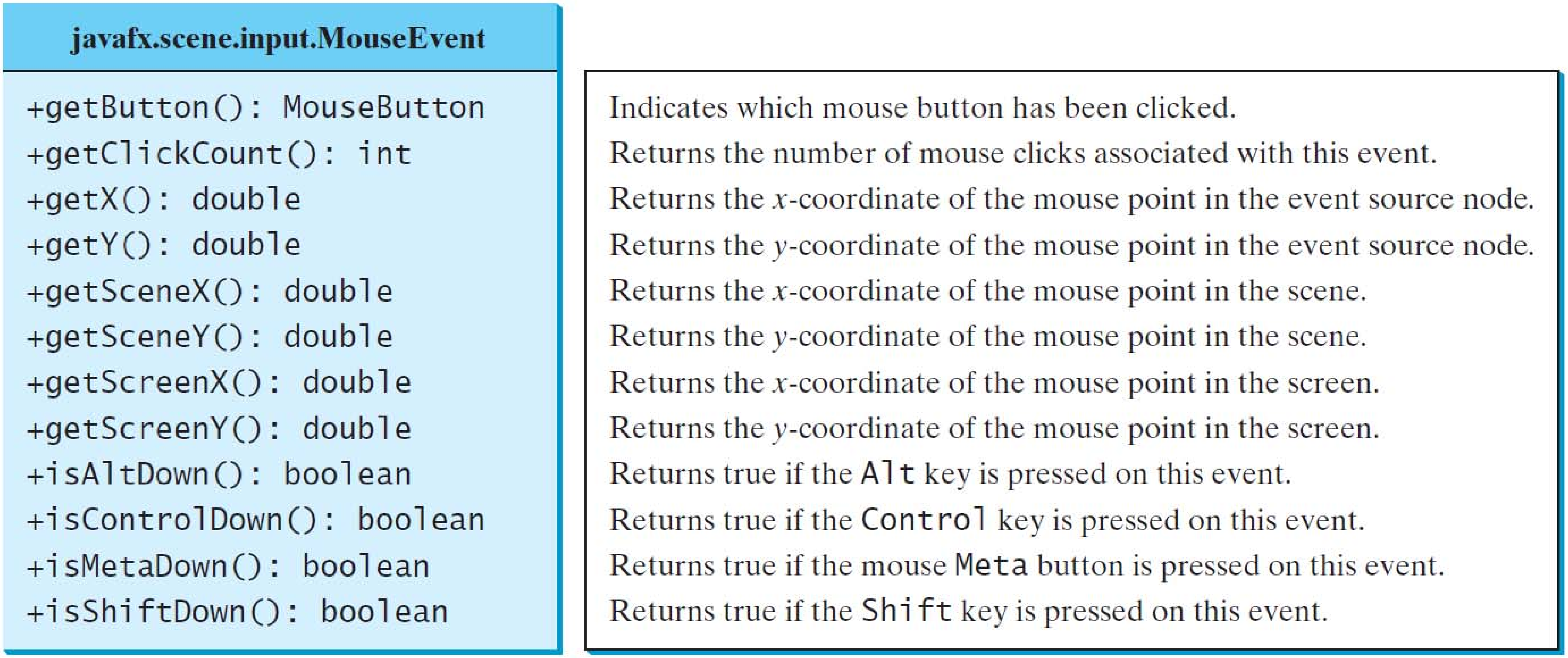
- Four constants—**PRIMARY**, **SECONDARY**, **MIDDLE**, and **NONE**—are defined in the MouseButton enumerator to indicate the left, right, middle, and none mouse buttons.

- We can use the **getButton**() method to detect which button is pressed. For example,

```
if (e.getButton() == MouseButton.SECONDARY)
```

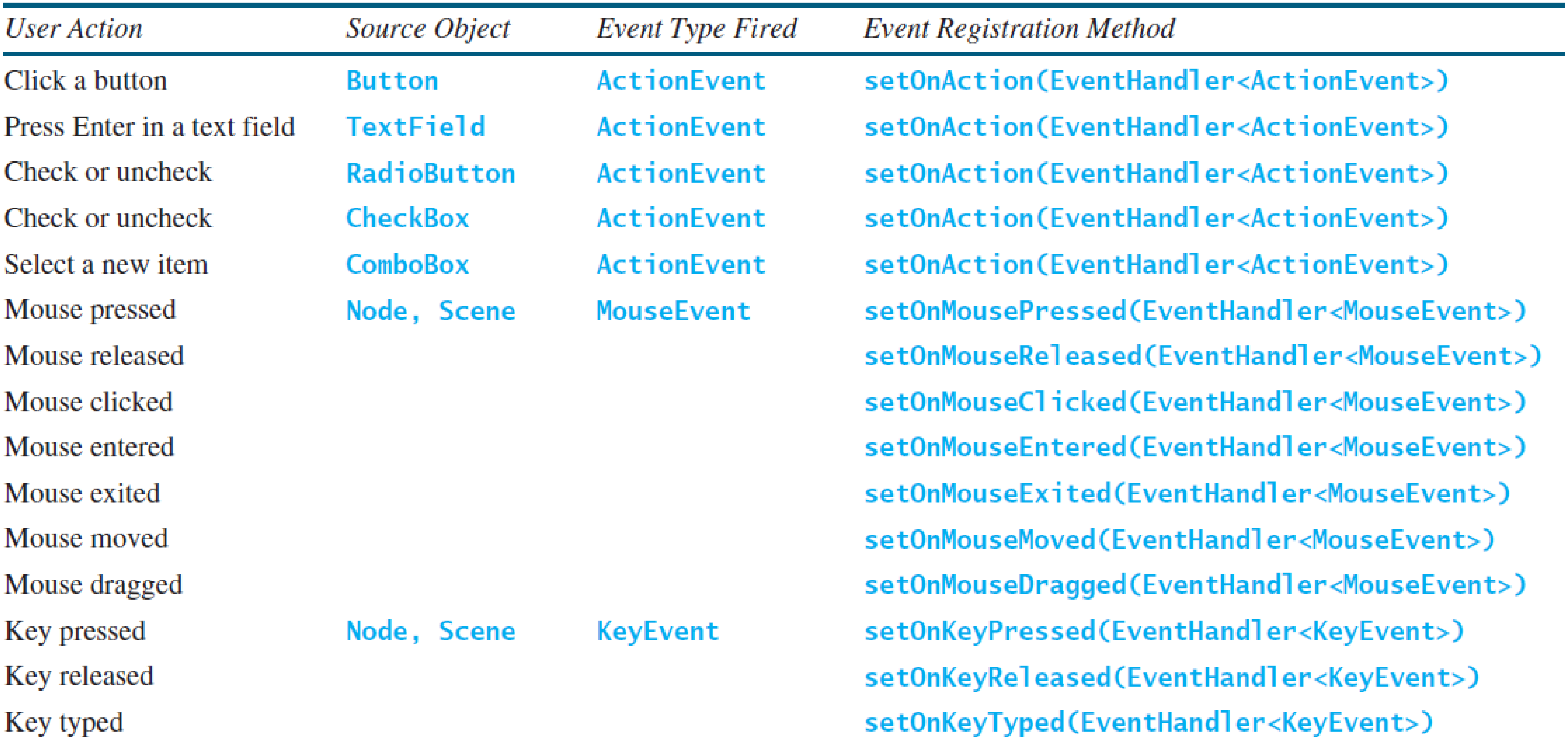
checks that the right mouse button was pressed.

The `MouseEvent` Class



javafx.scene.input.MouseEvent	
`+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.

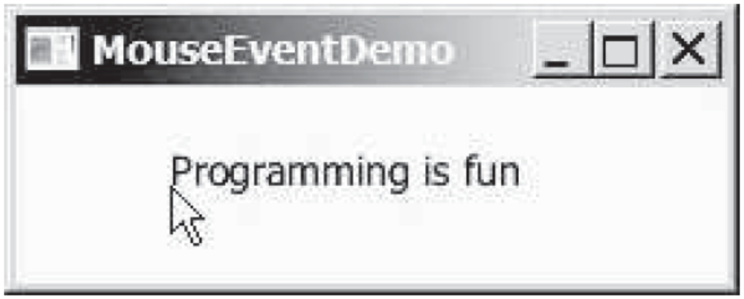
Mouse Events



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>)`

MouseEvent Example

```
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
  }
}
```

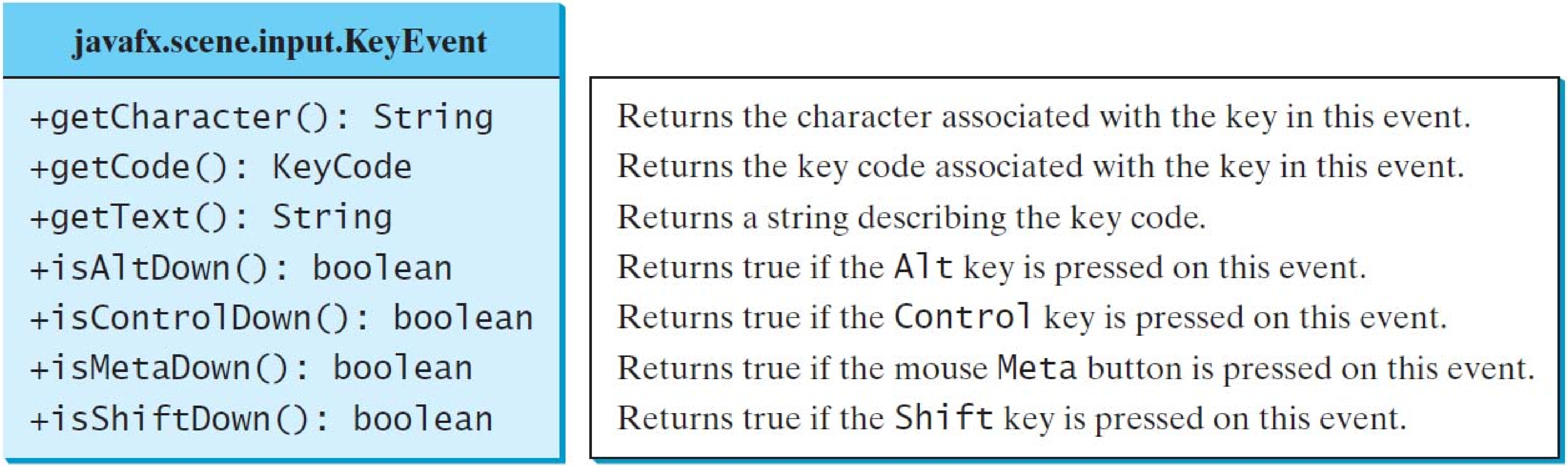


Programming is fun

Key Events

- A **KeyEvent** is fired whenever a key is pressed, released, or typed on a node or a scene.
- Key events enable the use of the keys to perform actions or to get input from the keyboard.
- The **KeyEvent** object describes the type of the event (key pressed, key released, or key typed) and the value of the key.

The KeyEvent Class

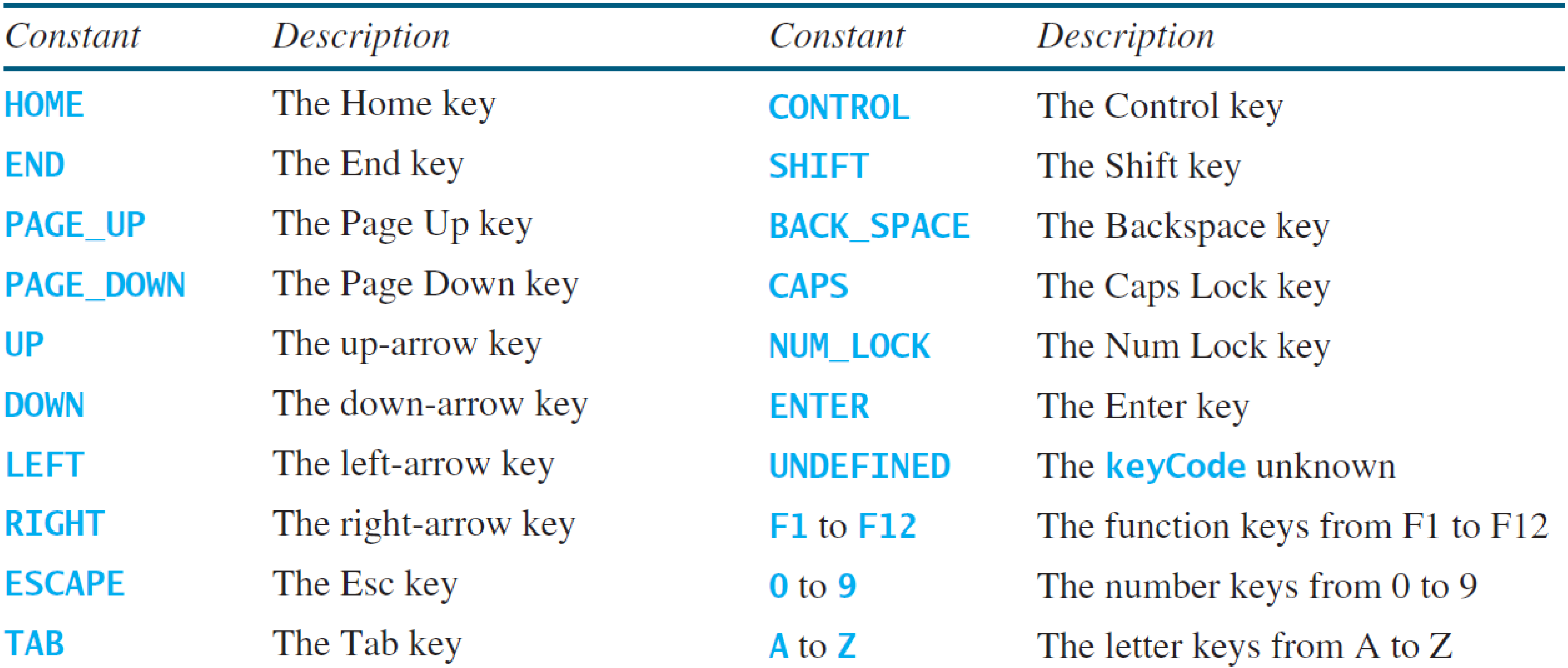


javafx.scene.input.KeyEvent	
+getCharacter(): String	Returns the character associated with the key in this event.
+getCode(): KeyCode	Returns the key code associated with the key in this event.
+getText(): String	Returns a string describing the key code.
+isAltDown(): boolean	Returns true if the Alt 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.

Key Events

- Every key event has an associated code that is returned by the **getCode**() method in **KeyEvent**.
- The *key codes* are constants defined in the enumerator **KeyCode**.
- For the *key-pressed* and *key-released* events, **getCode**() returns the value as defined in the table, **getText**() returns a string that describes the key code, and **getCharacter**() returns an empty string.
- For the *key-typed* event, **getCode**() returns **UNDEFINED** and **getCharacter**() returns the Unicode character or a sequence of characters associated with the *key-typed* event.

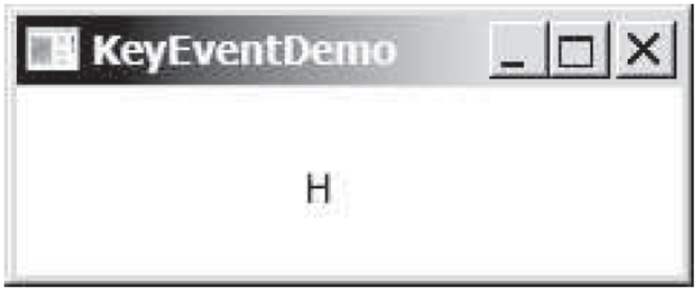
The KeyCode Constants



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

Key Events Example 1

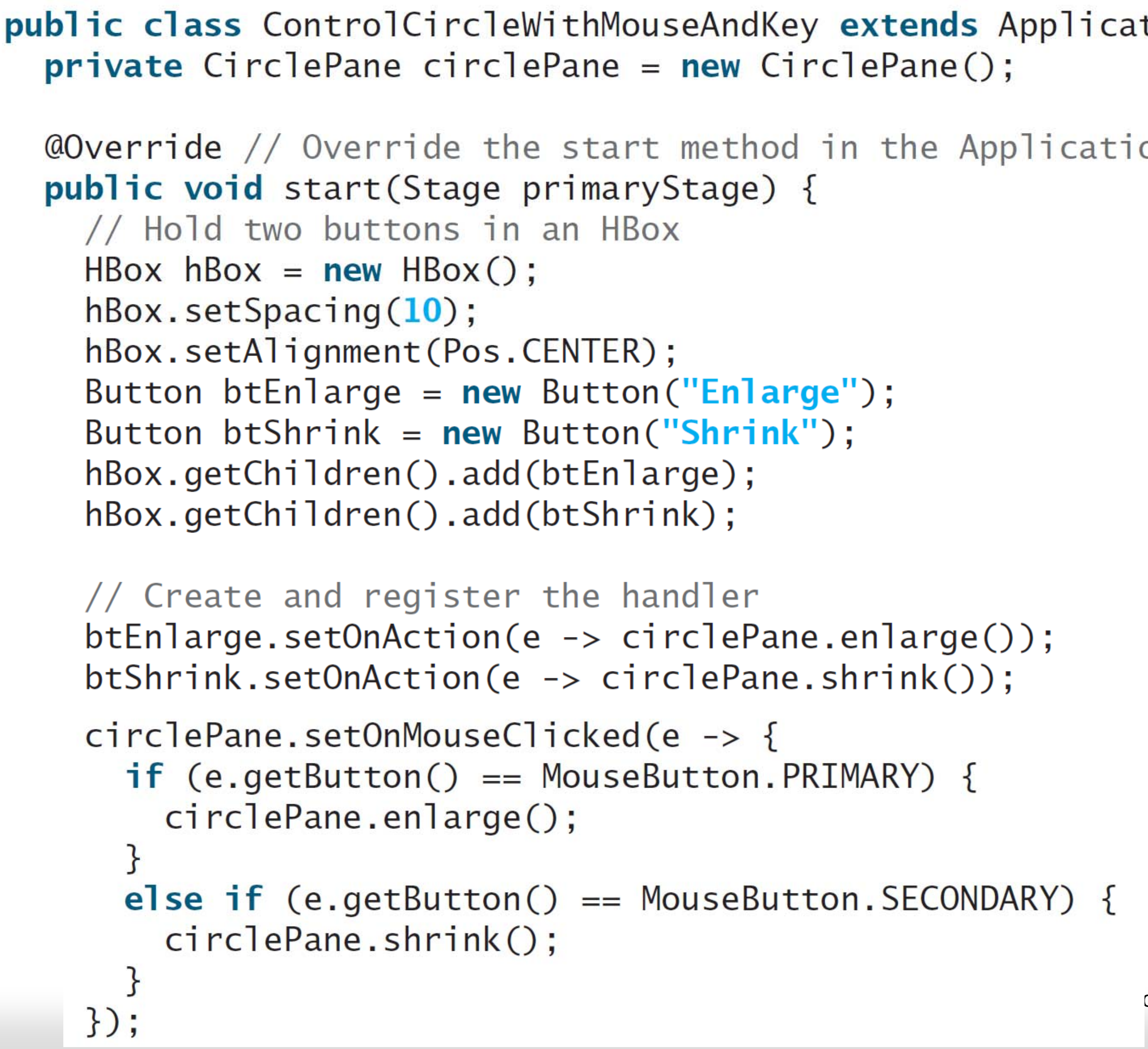
```
public class KeyEventDemo extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a pane and set its properties
    Pane pane = new Pane();
    Text text = new Text(20, 20, "A");

    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());
  }
});
```



Key Events Example 1

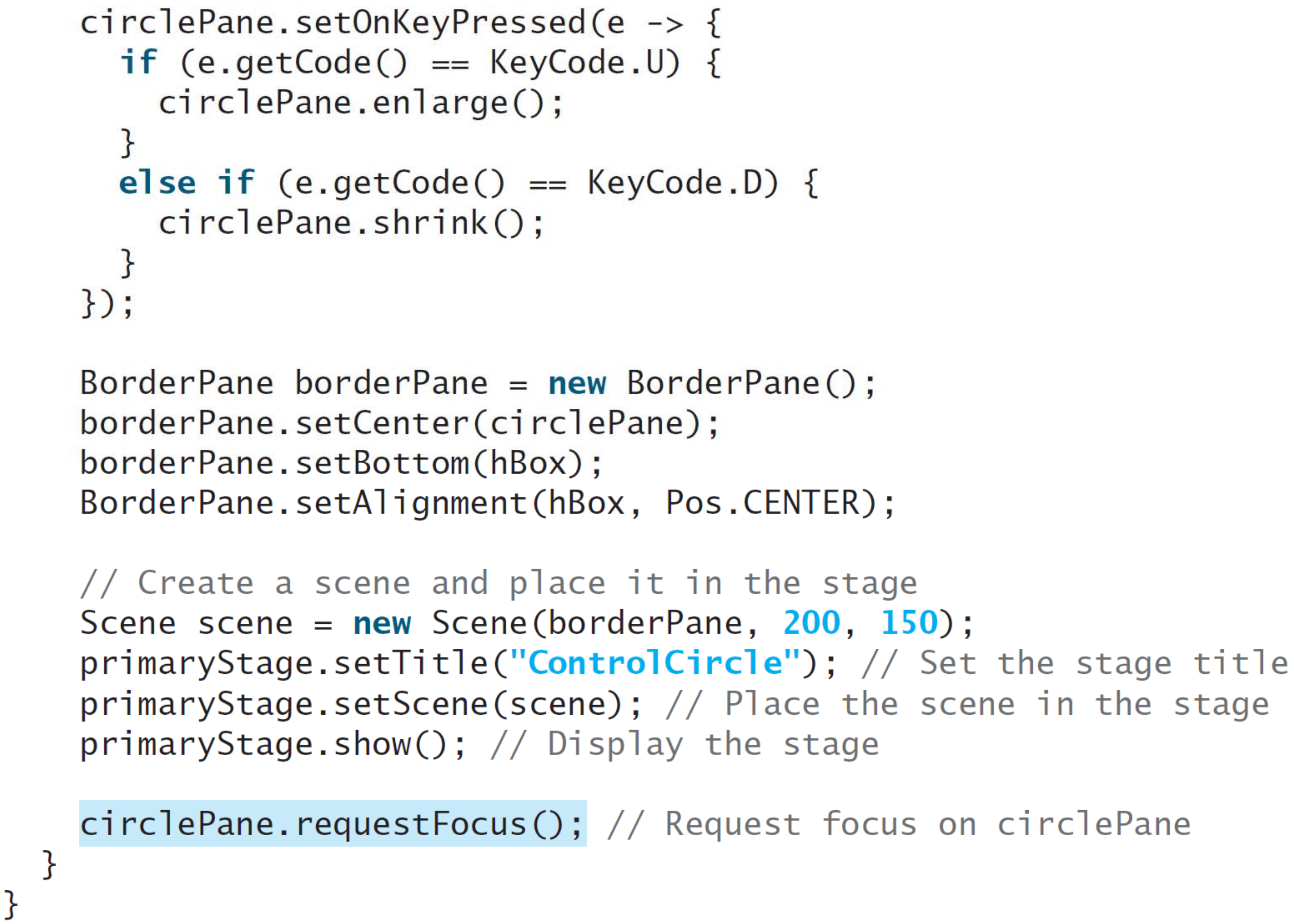
- In a *switch* statement for an **enum** type value, the *cases* are for the *enum* constants. The constants are unqualified.
- For example, using **KeyCode.DOWN** in the *case* clause will be wrong and produce an error.
- Only a focused node can receive **KeyEvent**. Invoking **requestFocus()** on **text** enables **text** to receive key input. This method must be invoked after the stage is displayed.

Key Events Example 2



```
public class ControlCircleWithMouseAndKey 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(e -> circlePane.enlarge());
    btShrink.setOnAction(e -> circlePane.shrink());

    circlePane.setOnMouseClicked(e -> {
      if (e.getButton() == MouseButton.PRIMARY) {
        circlePane.enlarge();
      }
      else if (e.getButton() == MouseButton.SECONDARY) {
        circlePane.shrink();
      }
    });
```

Key Events Example 2



```
    circlePane.setOnKeyPressed(e -> {
      if (e.getCode() == KeyCode.U) {
        circlePane.enlarge();
      }
      else if (e.getCode() == KeyCode.D) {
        circlePane.shrink();
      }
    });

    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

    circlePane.requestFocus(); // Request focus on circlePane
  }
}
```