Event Handling in JavaFX

Event-Driven Programming

Introduction

- This section will look at handling events in JavaFX applications.
- A number of important classes, interfaces and idioms/patterns for handling events will be considered.
- Note that we will consider just a few selected events for selected GUI components for illustration purposes. The principles are the same regardless of which events and components we want to use, though.
  - See the JavaFX API documentation for more detail.

Event Handling

- Event handling concerns those actions that are taken by a program in response to a given event.
- Event examples:
  - Button is clicked
  - Text box is changed
  - Key is pressed / released
  - Scroll bar is moved
  - Mouse is moved / dragged / clicked etc.
- Events can be user generated (i.e. user does something) or system generated (i.e. something happens in the background).
  - We only handle / respond to events we are interested in!
- We will concentrate on user generated events for JavaFX components.

Event Sources and Handlers

- Two key concepts for event-driven applications are:
  - Event sources
  - Event handlers (a.k.a. event listeners)
- An event source is an object which gives rise to the event (e.g. a button).
  - Different types of sources give rise to different types of events.
- An event handler is an object that is notified of and responds to an event.
  - Different handlers are used for different types of events.

Associating Event Handlers With Event Sources

- In order to be notified of any given event, the event handler must be registered with any event source(s) it is to respond to.
  - Note that a single event handler can be used for multiple event sources.
  - Note also that a single source can have many handlers.
- Event sources essentially “remember” these event handlers, and will call the relevant event handling method of the handler when the relevant event occurs in order to notify it.

General Steps to Event Handling in Java Applications

1. Identify the event source (e.g. button).
2. Determine which event type you are interested in that can be caused by the source (e.g. click = ActionEvent, mouse entered = MouseEvent, etc.).
3. Create an event hander object that will accept that type of event (e.g. ActionEvent, MouseEvent, etc.).
   - Generally, the event handler class will implement a Java interface that provides for that event type. The code to run when the event occurs is specified in a specific method required of the interface.
4. Register the event handler object with the event source.

Event Handling in JavaFX

- Different GUI components in JavaFX cause different types of events. These components are our event sources.
  - Button, CheckBox, TextField, ListView, Label, etc.
  - Note that all Node objects cause some events! (e.g. mouse entering, dragging, key presses, etc.)
- All event classes in JavaFX are (directly or indirectly) subclasses of Event. These include:
  - ActionEvent, MouseEvent, KeyEvent, etc.
- In order to handle events of interest (ignore other events!) we create an event handler class that implements the EventHandler interface for the required event type.
- An instance of the event handler class can then be registered with the event source using the setEventHandler() method or a convenience method.

EventHandler Interface

- The EventHandler interface is used for (i.e. must be implemented by) all event handler objects in JavaFX.
- To use the EventHandler interface:
  - The event class type we want to handle is specified via the generic type (pointy brackets!) next to EventHandler in the implementing class header definition.
  - We then provide the method handle() required by the EventHander interface, specifying its single parameter type as the same as the event class type being handled.

Continued…

- For instance, a class to handle events of type ActionEvent might look as follows:

```
package eventexamples;

import javafx.event.*;

public class MyActionHandler implements EventHandler<ActionEvent> {

@Override
public void handle(ActionEvent e) {
    // TODO Auto-generated method stub

    //Add code to respond to event here
    System.out.println("Hey, something just happened!");

}

}
```

Note that these (i.e. generic and parameter types) must match! Change for whatever event type you want to handle (e.g. MouseEvent).

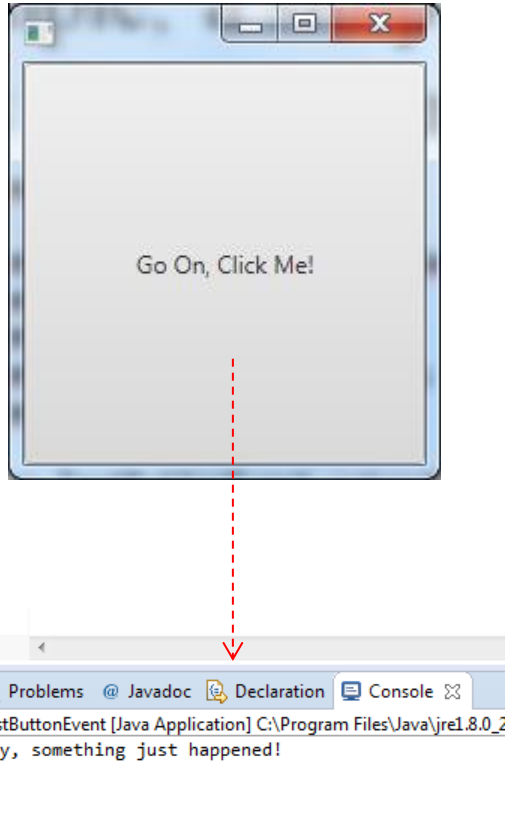
Using the Event Handler

- As previously noted, we have to register the event handler with an event source in order to use it.
- We can use the addEventHandler() method to do this.
- addEventHandler() requires two arguments:
  - The first argument specifies the EventType associated with the event. This is generally available as a constant (public static final) in the relevant event class.
    - Examples: ActionEvent.ACTION, MouseEvent.MOUSE_PRESSED, KeyEvent.KEY_TYPED, etc.
    - The EventType constant ANY is available for all event classes too.
  - The second argument is a reference to the event handler object.
    - That is, an object whose class implements the EventHandler interface for the event type required.

Continued…

- For example, a simple one button application that uses the previous ActionEvent handler class:

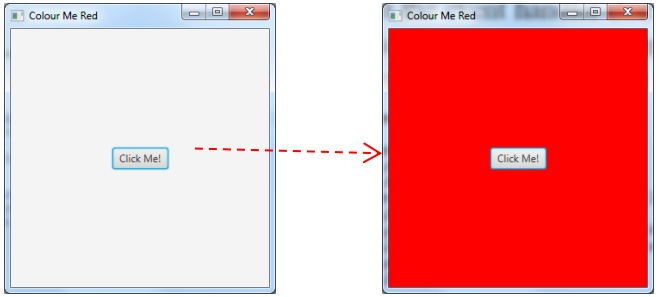
```
package eventexamples;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.stage.Stage;

public class TestButtonEvent extends Application {

@Override
      public void start(Stage ps) {
      // TODO Auto-generated method stub
      Button b=new Button("Go On, Click Me!");

      MyActionHandler ah=new MyActionHandler();
      b.addEventHandler(ActionEvent.ACTION, ah);

      ps.setScene(new Scene(b,200,200));
      ps.show();
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}
}
```



Continued…

- Note that the event handler does not have to be a separate class. Any class that implements EventHandler can be the event handler regardless of what else it is, contains, or is doing:

```
package eventexamples;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.*;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class ColourMeRed extends Application
          implements EventHandler<ActionEvent> {

private FlowPane fp=new FlowPane();

@Override
public void start(Stage primaryStage) {
      // TODO Auto-generated method stub
      Button b=new Button("Click Me!");
      fp.getChildren().add(b);
      fp.setAlignment(Pos.CENTER);

    b.addEventHandler(ActionEvent.ACTION, this);
      primaryStage.setScene(new Scene(fp,300,300));
      primaryStage.setTitle("Colour Me Red");
      primaryStage.show();
}

@Override
public void handle(ActionEvent e) {
      // TODO Auto-generated method stub
      fp.setBackground(new Background(new
      BackgroundFill(Color.RED,CornerRadii.EMPTY,
            Insets.EMPTY)));
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}
}
```



Determining Event Sources

- The single argument to handle() is a reference to the event object, which describes the event that occurred:

```
public void handle(ActionEvent e) { … }
```

- This reference allows us to easily determine the source of the event using the getSource() method regardless of the event type.

```
Object x = e.getSource();
```

  - Note that we can typecast the Object returned by getSource() to the required class (e.g. Button) as required.

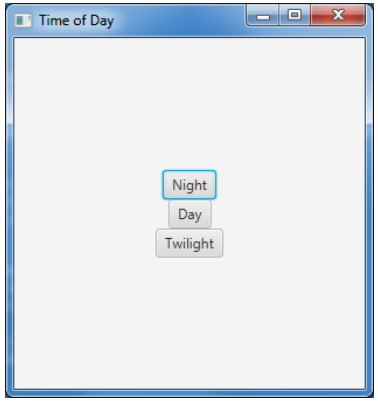
Continued…

- Being able to determine the event source allows multiple sources to use the same event handler object. Example:

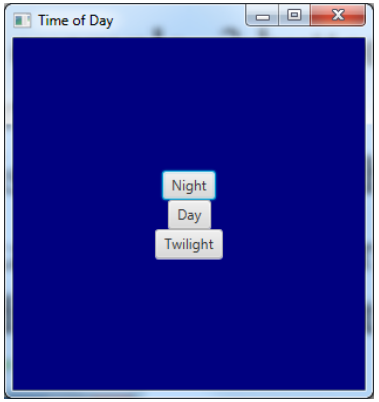
```
package eventexamples;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.*;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class NightAndDay extends Application
            implements EventHandler<ActionEvent> {

private VBox root=new VBox();
private Button night=new Button("Night"), day=new
Button("Day"), twilight=new Button("Twilight");

@Override
public void start(Stage primaryStage) {
      // TODO Auto-generated method stub
      root.getChildren().addAll(night,day,twilight);
      root.setAlignment(Pos.CENTER);

    night.addEventHandler(ActionEvent.ACTION, this);
    day.addEventHandler(ActionEvent.ACTION, this);
    twilight.addEventHandler(ActionEvent.ACTION, this);
      primaryStage.setScene(new Scene(root,300,300));
      primaryStage.setTitle("Time of Day");
      primaryStage.show();
}

@Override
public void handle(ActionEvent e) {
// TODO Auto-generated method stub
      Color dayColor;

    if(e.getSource()==night) dayColor=Color.NAVY;
    else if(e.getSource()==day)
          dayColor=Color.LIGHTSKYBLUE;
    else dayColor=Color.SLATEBLUE;

      root.setBackground(new Background(new
      BackgroundFill(dayColor,CornerRadii.EMPTY,
            Insets.EMPTY)));
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}
}
```

Continued…

- In this example, 3 buttons share the same event handler. Which one is the source of the event is determined in the handle() method.
- The example just colours the root pane depending on which button is clicked.



Initial



Night

Day

Twilight

Determining Event Type

- A single event handler can also deal with different types of events (e.g. both ActionEvents and MouseEvents).
- In order to do this, use the generic type <Event> when implementing the EventHandler interface, and use the method getEventType() in handle() on the event reference argument to determine the event type.
  - Note that it is not possible for a class to implement EventHandler multiple times for different event types instead due to “type erasure” on generic types at compile time.

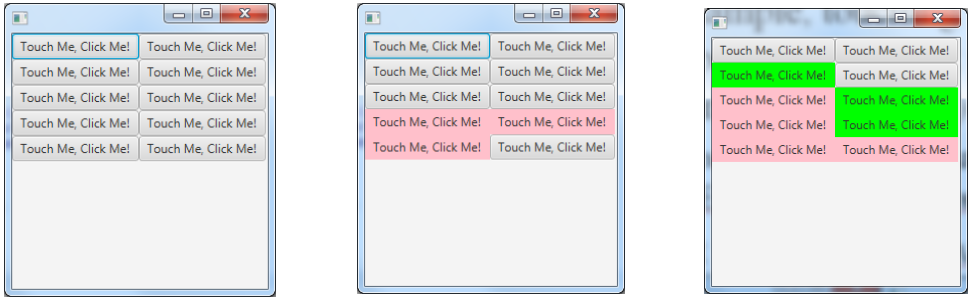
Continued…

- Example: Event handler class dealing with both ActionEvent and MouseEvent events:

```
package eventexamples;

import javafx.application.Application;
import javafx.event.*;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class EnterAndClick extends Application
            implements EventHandler<Event> {

@Override
public void start(Stage ps)  {
      // TODO Auto-generated method stub
      FlowPane fp=new FlowPane();

      for(int i=0;i<10;i++)
      {
      Button b=new Button("Touch Me, Click Me!");

      b.addEventHandler(ActionEvent.ACTION, this);
      b.addEventHandler(MouseEvent.MOUSE_ENTERED, this);

      fp.getChildren().add(b);
      }
      ps.setScene(new Scene(fp,300,250));
      ps.show();
}

@Override
public void handle( Event e ) {
      // TODO Auto-generated method stub
      Region p=(Region)e.getSource();

      if(e.getEventType()==MouseEvent.MOUSE_ENTERED)
            p.setBackground(new Background(new
            BackgroundFill(Color.PINK,CornerRadii.EMPTY,
            Insets.EMPTY)));
      if(e.getEventType()==ActionEvent.ACTION)
            p.setBackground(new Background(new
            BackgroundFill(Color.LIME,CornerRadii.EMPTY,
            Insets.EMPTY)));
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}
}
```

Continued…

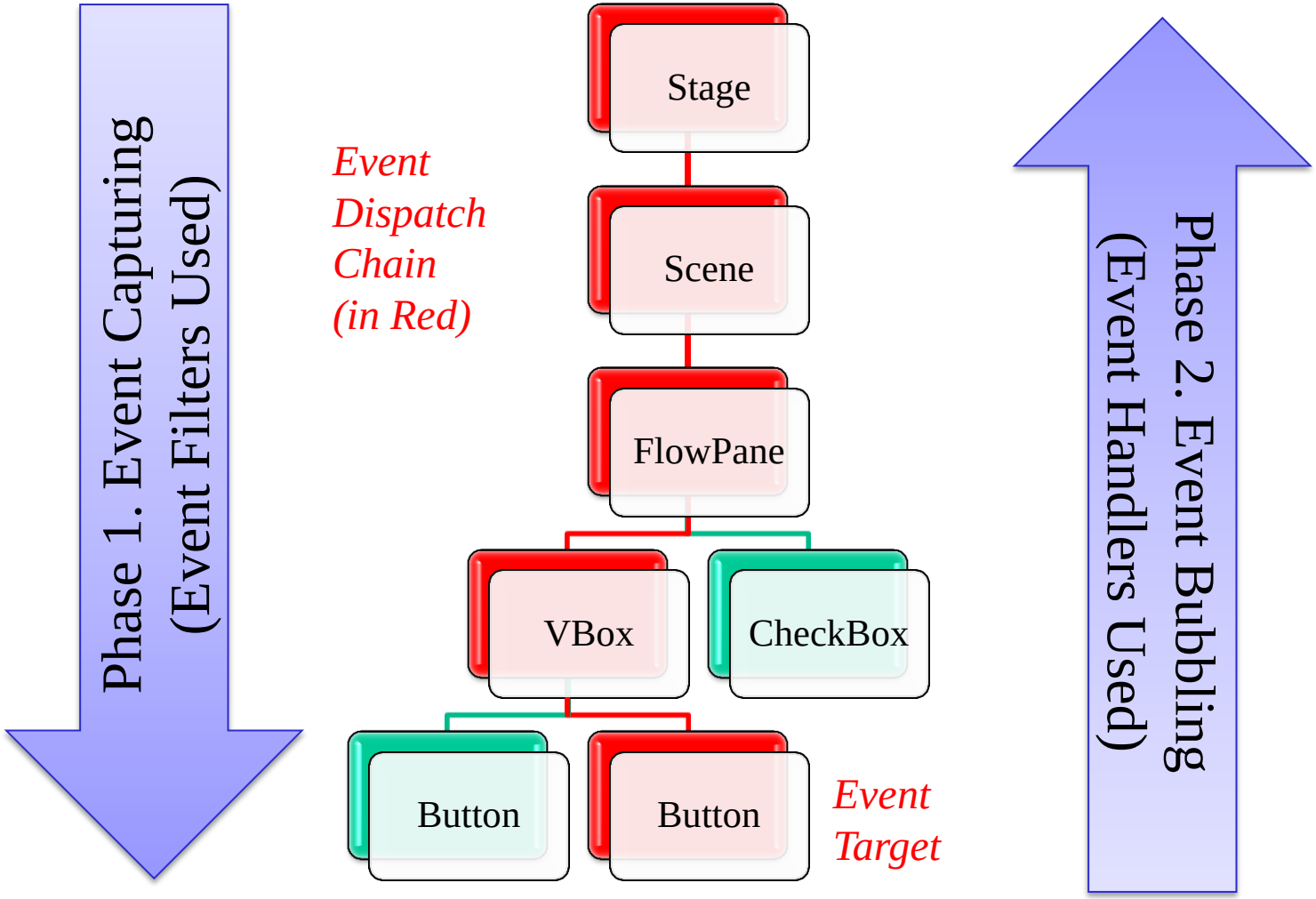
- In this example, touching (mouse entered MouseEvent) any component colours it pink, and clicking any component (ActionEvent) colours it green.
  - Provided the component has added the handler for MouseEvents and/or ActionEvents as appropriate.
- Note that we could use Event.ANY in the addEventHandler() method so that all events on the component would go to the handler (there will be lots of them though!).



The Event Dispatch Chain

- The event dispatch chain is the route from the Stage through the scene graph to the target Node (e.g. Button) of the event.
- When an event occurs:
  - The event is passed from the Stage down the scene graph towards the target Node. Event filters can be used to process events at any Node (e.g. consume if required). The event eventually reaches the target (if not consumed by a filter). This is the "event capturing phase".
    - Note that addEventFilter() works in the same way as addEventHandler().
  - Once the event reaches the target Node (and processed as required), it is passed back up the chain towards the Stage. Event handlers (as already seen) handle the events. This is the "event bubbling phase".
- A single event may therefore be processed by numerous Nodes during event capturing/bubbling.
- We can prevent an event from continuing up/down the chain by consuming it during bubbling/capturing as appropriate.
  - consume() method.

Continued…


Phase 1. Event Capturing
(Event Filters Used)
Event
Dispatch
Chain
(in Red)
Stage
Scene
FlowPane
VBox
CheckBox
Button
Button
Event
Target
Phase 2. Event Bubbling
(Event Handlers Used)

Convenience Methods

- We have seen how to implement event handling directly using the addEventHandler() method.
  - Note that addEventHandler() requires us to specify the event type along with the event handler.
- Another way we can do this (a *little* more simply) is via "convenience methods" provided by many JavaFX components.
- Convenience methods are essentially setter methods that set special event handling properties in JavaFX components to automatically register and use user-specified event handlers.

Continued…

- Convenience methods have the form:

  setOnXXX(EventHandler) where XXX specifies the type of event to be handled.

- One example convenience method is setOnAction() which can be used for ActionEvent handling. Using this method we can use:

  mycomponent.setOnAction(myActionEventHandler);

- In place of:

  mycomponent.setEventHandler(ActionEvent.ACTION, myActionEventHandler);

- setOnAction() sets the component's onAction property to refer to the single specified ActionEvent handler.
  - In general, setOnXXX() sets the onXXX property.

Continued…

- Other convenience methods work in the same way, including:
  - setOnMouseMoved()
  - setOnMousePressed()
  - setOnMouseDragged()
  - setOnDragOver()
  - setOnDragEntered()
  - setOnKeyPressed()
  - setOnKeyReleased()
  - setOnContextMenuRequested()
  - setOnSwipeLeft()
  - setOnTouchMoved()
  - setOnZoom()
  - setOnRotate()
  - setOnRotationStarted()
  - And others!
- Note that some convenience methods may only be available for certain Node components.
- Note that some events may have certain requirements in order to work/occur e.g. a touch screen for gesture events (swipe, zoom, etc.).

Anonymous Classes and Event Handling

- The use of anonymous classes for event handling is a common idiom in Java programming.
- Anonymous classes are local unnamed classes that we define and instantiate at the same time as an expression for a single use.
- The syntax of an anonymous class expression is like a constructor invocation, except that there is a class definition contained in a block of code.

Continued…

- An anonymous class can be based on either an interface or class.
  - Interfaces are implemented, and classes are extended as usual.
- The syntax of an anonymous class expression is:

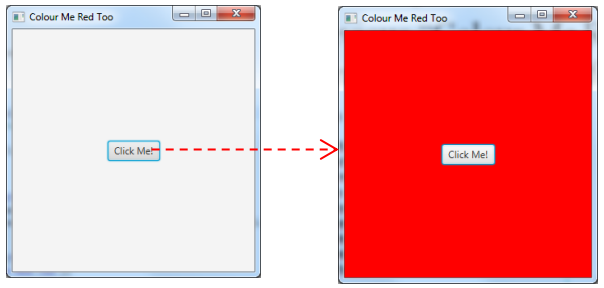
```
new <ClassOrInterface> (<ClassConstructorArgs>) {
<MethodDefinitionsEtc>   }
```

- An anonymous class can access members of its enclosing class, as well as final (or effectively final) local scoped variables.

Anonymous Class Example

- The previous "Colour Me Red" example implemented using an anonymous class for the event handling:

```
package eventexamples;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.*;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.stage.Stage;

public class ColourMeRed2 extends Application {

private FlowPane fp=new FlowPane();

@Override
public void start(Stage primaryStage) {
      // TODO Auto-generated method stub

      Button b=new Button("Click Me!");
      fp.getChildren().add(b);
      fp.setAlignment(Pos.CENTER);
      b.addEventHandler(ActionEvent.ACTION,
            new EventHandler<ActionEvent>() {
            public void handle(ActionEvent e) {
            fp.setBackground(new Background(new
            BackgroundFill(Color.RED,CornerRadii.EMPTY,
            Insets.EMPTY)));
      }}
      );

      primaryStage.setScene(new Scene(fp,300,300));
      primaryStage.setTitle("Colour Me Red Too");
      primaryStage.show();
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}
}
```



Continued…

- If we wanted to use a convenience method as well as an anonymous class we could have used:

```
b.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
    fp.setBackground(new Background(new
    BackgroundFill(Color.RED,CornerRadii.EMPTY,
    Insets.EMPTY)));
});
```

- It is also possible to declare an anonymous class instance and assign it to a reference variable for future use. However, the inline form seen here is very common in practice.

Lambda Expressions

- Lambda expressions are a new and important feature of Java 8.
  - Some other functional programming languages (e.g. Common Lisp) have supported lambda expressions for decades.
- The general principle of lambda expressions is that of “code as data” and “data as code”.
  - A lambda expression is “data” but can be executed as “code” in a functional sense. It is therefore simultaneously both code and data.
- Lambda expressions in Java are essentially blocks of code that can be e.g. passed to methods and executed when required.
  - Lambda expressions can include parameter variables.
  - Lambda expressions are still represented as objects in first-class OO Java.

Continued…

- Lambda expressions in Java can be used to concisely implement a functional interface.
  - A functional interface is an interface that contains one abstract method.
- Lambda expressions thus enable us to express instances of single-method classes more compactly/concisely than via anonymous classes.
- Lambda expressions are a convenient and powerful means of dealing with events in Java 8
  - Note that lambda expressions are not limited to event handling. A number of general purpose standard functional interfaces (e.g. Predicate<T>, which has a single method test()) are provided in Java.

Continued…

- The format of lambda expressions is as follows:

```
(<MethodParameters>) -> { <Code> }
```

- The arrow (`->`) operator is key to lambda expressions!
- As the interface being implemented only has a single method, the name and return type of the method do not need to be specified.
- Method parameter notes:
  - No parameters: use empty brackets.

```
() -> { … }
```

  - One parameter: specify type and choose parameter name.

```
(ActionEvent e) -> { … }
```

  - However, parameter types can generally be inferred from the class/method definitions, so we do not need to specify the types.

```
(e) -> { … }
```

Continued…

- When we only have a single parameter (with no specified type), we do not need to use the parentheses either.

```
e -> { … }
```

- When we have multiple parameters, use the parentheses and comma separate them. Again, types are optional.

```
(String a, int b) -> { … }  //or
(a, b) -> { … }
//for a functional interface whose single method (name is
//immaterial) takes a String and an int as parameters.
```

- Method code notes:
  - Can specify numerous (semicolon terminated) statements inside the braces { }.
  - Can leave out the braces if only a single statement is specified.
    - Note that the result of a single statement is automatically returned.

Lambda Expression Example

- Consider the anonymous class from the previous example (using the setOnAction() convenience method for conciseness):

```
b.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {
    fp.setBackground(new Background(new
    BackgroundFill(Color.RED,CornerRadii.EMPTY,
    Insets.EMPTY)));
});
```

- Note how much of this code is defining / implementing the anonymous class as opposed to actually handling the event.

Continued…

- Rewriting this code to use a lambda expression instead:

```
b.setOnAction(e -> fp.setBackground(new Background(
        new BackgroundFill(Color.RED,CornerRadii.EMPTY,
        Insets.EMPTY)))
);
```

- Note how we no longer have to provide:

```
new EventHandler<ActionEvent>() {
    public void handle(ActionEvent …
```

- Note also how Java already knows that the method to invoke is handle() and how the type of the method parameter (e) is ActionEvent.
  - setOnAction() takes an EventHandler<ActionEvent> argument!
- This is the big advantage of using lambda expressions for event handling: conciseness!

Method References

- Another new addition to Java 8, related to lambda expressions, is “method referencing”, which allow us to pass a reference to a named method as a parameter in place of a functional interface type.
  - The referenced method simply has to have the same form (parameters and return type) as the abstract method in the functional interface. It can have a different name, though.
- The double colon operator (::) is used for method references, which can refer to static or non-static methods as usual:
  - ClassName::myStaticMethodRef
  - myObject::myNonStaticMethodRef
  - Can also use this::someMethod and super::someMethod

Continued…

- For example, if we wanted to use a method reference as the setOnAction() method argument, we simply have to make sure that the method has the same form as the handle() method in the EventHandler functional interface.
- As setOnAction() requires a parameter type EventHandler<ActionEvent>, and this generic type propagates to the parameter of handle(), and setOnAction() has a void return type, our method simply has to have the form:

```
void whatever(ActionEvent e) { … }
```

Continued…

- For instance, code that would work in the "Colour Me Red" example (as another / alternative approach to event handling):

```
//Put inside ColourMeRed class as a new method
public void goRed(ActionEvent e) {
    fp.setBackground(new Background(new
    BackgroundFill(Color.RED,CornerRadii.EMPTY,Insets.EMPTY)));

}
            .
            .

b.setOnAction(this::goRed); //Put inside the constructor
            .
            .
```

Event Handling with FXML

- Where the user interface has been designed with FXML (and possibly styled using CSS) it is easy to add event handlers to any component.
- This can be done by:
  - Setting the selector ID property for the component in the FXML code. Example:

```
<Button id="my-funky-button" … > …
```

  - Retrieving an object reference to the required component using the lookup() method on either the Scene or root component, passing it the selector ID that was specified in the FXML.

```
Button mb = (Button)rootNode.lookup("#my-funky-button");
```

    - *Note: hash (#) is used before the selector ID in lookup()*

Continued…

- Registering an appropriate event handler with the component using the object reference. This can be done using addEventHandler() or a convenience method as desired. The handler can be specified as a regular class, anonymous class, using a lambda expression, etc. as previously seen.

- For example (using convenience method and lambda expression):

```
mb.setOnAction(e -> doFunkyStuff());
```

- One statement example:

```
((Button)rootNode.lookup("#my-funky-button")).setOnAction(e ->
doFunkyStuff());
```

Continued…

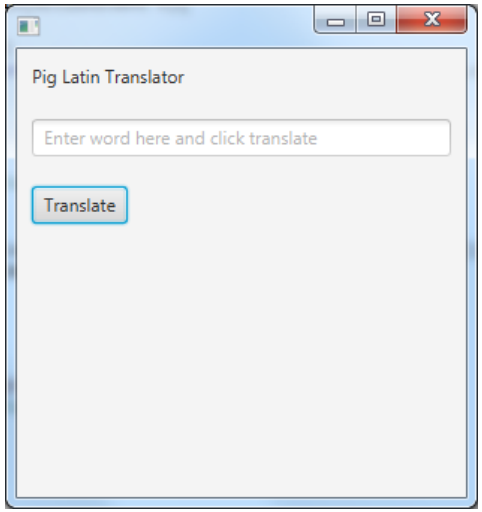
- An alternative way to link FXML-specified components to event handling behaviour in the Java code is to specify Java methods to invoke for event properties in the FXML itself.
- Notes:
  - We have to set the fx:controller property in the FXML (root node attribute) to refer to the Java class that contains the event handling method(s) being used therein.
  - The event handling methods in the Java code should be implemented in the specified Java class (and marked/preceded with the annotation @FXML unless they are public). The methods can (optionally) take a parameter of the type of event to be handled (e.g. ActionEvent), which will provide information on events when they occur.

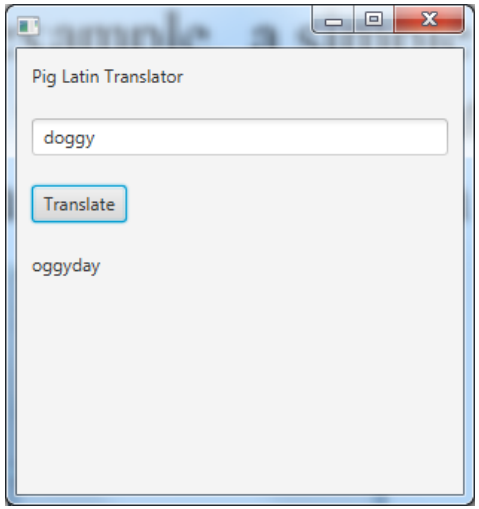
Continued…

  - We can also get automatic references to FXML components by setting their fx:id property in the FXML code, and then create a suitable object reference with the same name in Java (preceded by @FXML unless they are public). This is an alternative to using the lookup() method as previously seen.
  - We then set the onXXX property for the event source / component (e.g. Button) where XXX is the event we are interested in (e.g. onAction, onMouseEntered, etc.; see notes on convenience methods for more). The property should be set to the required event handling method name preceded by a hash (#).
- Note that Scene Builder allows us to set these various properties for the FXML code in property fields, or we can edit the FXML code manually.

Continued…

- For example, an application that asks for a word and translates it to simple Pig Latin when the button is clicked:





  - Note: To convert a word to Pig Latin: move the first letter of the word to the end and add "ay" (e.g. hello = ellohay)

Continued…

- The FXML code (PigLatin.fxml):

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.shape.*?>
<?import java.lang.*?>
<?import javafx.scene.layout.*?>

<VBox maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-Infinity"
minWidth="-Infinity" prefHeight="300.0" prefWidth="300.0" spacing="20.0"
xmlns="http://javafx.com/javafx/8" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="eventexamples.PigLatinTranslator">
    <children>
        <Label text="Pig Latin Translator" />
        <TextField fx:id="theword" promptText="Enter word here and click
            translate" />
        <Button mnemonicParsing="false" onAction="#translateToPigLatin"
            text="Translate" />
        <Label fx:id="thetranslation" />
    </children>
    <padding>
        <Insets bottom="10.0" left="10.0" right="10.0" top="10.0" />
    </padding>
</VBox>
```

Continued…

- The Java code (PigLatinTranslator.java):

```
package eventexamples;

import java.io.IOException;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.control.TextField;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

public class PigLatinTranslator extends
Application {

@FXML private TextField theword;
@FXML private Label thetranslation;

@FXML private void
translateToPigLatin(ActionEvent e){
      if(theword.getText().length()>0)
      thetranslation.setText(theword.getText().
      substring(1)+theword.getText().charAt(0)+
      "ay");
}
```

```
@Override
public void start(Stage ps) {
      // TODO Auto-generated method stub
      try{
      VBox
      root=(VBox)FXMLLoader.load(getClass().get
      Resource("PigLatin.fxml"));
      ps.setScene(new Scene(root,300,300));
      ps.show();
      }catch(IOException e){}
}

public static void main(String[] args) {
      // TODO Auto-generated method stub
      launch(args);
}

}
```

Note that the @FXML annotation does not have to be used on public attributes or methods.

Scripting in FXML

- It is possible to handle events in a suitable scripting language locally within the FXML code itself if desired (i.e. separate to the Java code).
  - Can also script for dynamic layout, etc.
- Any language that provides a JSR 223-compatible scripting engine could be used for this purpose.
  - JavaScript, for example.
- We will not consider this scripting option in this module, suffice to say that it is possible and straightforward.

Swing Overview

- Swing (and AWT) event handlers are implemented by various listener interfaces:
  - ActionListener, MouseListener, KeyListener, etc.
  - Different listeners have different methods.
  - Different listeners handle different events (e.g. ActionEvent, MouseEvent, ChangeEvent, etc.)
- Event handlers can be implemented using standalone classes, anonymous classes, etc.
  - Lambda expressions would also now work in Java 8.
- We add/register event listeners with components using the addXXXListener() method (XXX = Event type):
  - addActionListener, addMouseListener, etc.
  - Example: mybutton.addActionListener(myactionhandler);