Outline

• Animation
• Layout Panes
• Absolute Positioning
• Shapes

Animation – Change Over Time

• Suppose we have an alien shape we would like to animate (e.g., make him move across the screen)
• As in film and video animation, we can create apparent motion with many small changes in position
• If we move fast enough and in small enough increments, we get smooth motion
• Same goes for size, orientation, shape change, etc...
• How to orchestrate a sequence of incremental changes?
  • By coordinating them with a Timeline where change happens at defined instants

Introducing Timelines (1/2)

• In our use of KeyFrame, the KeyFrame(s) work with the Timeline to control the animation (e.g., the Duration of the KeyFrame and the number of repetitions). The actual changing of variables to affect the animation is done in EventHandler

Introducing Timelines (2/2)

• javafx.animation.Timeline is used to sequence one or more javafx.animation.KeyFrames, and optionally to run through them cyclically
  • Each KeyFrame lasts for its entire duration without making any changes, until its time interval ends and EventHandler is called
  • An EventHandler is used to define what should occur upon completion of each KeyFrame

Using JavaFX Timelines (1/2)

• KeyFrame and Timeline work together to control the animation, but our application’s EventHandler is the method that actually causes variables to change
Using JavaFX Timelines (2/2)

- We then pass our new KeyFrame into Timeline
- After we instantiate our Timeline, we must set its CycleCount property
  - This defines number of cycles in Animation
  - We will set cycle count to Animation.INDEFINITE, which will let Timeline run forever or until we explicitly stop it
- In order for Timeline to work, we must then call Timeline.play();

Another JavaFX App: Clock

- Simple example of a discrete (non-smooth) animation
- Specifications: App should display current date and time, updating every second
  - Useful classes:
    - java.util.Date
    - javafx.util.Duration
    - javafx.animation.KeyFrame
    - javafx.animation.Timeline

Process: Clock

1. Write App class that extends javafx.application.Application and implements start
2. Write a PaneOrganizer class that instantiates a node and returns it in a public getter method. Instantiate a Label and add it as a child. Factor out code for Timeline into its own method.
3. In our own setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler (defined later). Then instantiate Timeline, passing in our KeyFrame and play Timeline.
4. Write private inner TimeHandler class that implements EventHandler - it should know about a Label and update its text at every ActionEvent

Clock: App class

**Note: Exactly the same process as in ColorTextApp**

1. Instantiate a PaneOrganizer and store it in the local variable organizer
2. Instantiate a Scene, passing in organizer.getRoot(), and desired width and height of Scene
3. Set the Scene, set the Stage title, and show the Stage!
Process: Clock

1. Write an App class that extends javafx.application.Application and implements start.
2. Write a PaneOrganizer class that instantiates the root node and returns it as a public getRoot() method. Instantiate a Label and add it as a root node’s child. Factor out code for Timeline into its own method.
3. In our own setupTimeline(), instantiate a KeyFrame passing in a Function that doesn’t return a info about Label and update its text every ActionEvent.
4. Write a private inner Interface class that implements EventHandler (defined later). Then Instantiate a Timeline, passing in our Timeline, and play the Timeline.

Clock: Our PaneOrganizer Class

A. In the PaneOrganizer class constructor, instantiate a root VBox and set it as the return value of a public getRoot() method.

B. Instantiate a Label and add it to the list of the root node’s children.

C. Call setupTimeline(); will define this method next!!

Clock: PaneOrganizer class setupTimeline()

Within setupTimeline():

A. Instantiate a KeyFrame

B. Pass in info about Label and update its text every ActionEvent.

C. Note the info from Timeline’s handle method, which changes the label text before the next 1 second cycle starts.
Process: Clock

1. Write an App class that extends javafx.application.Application and implements start.
2. Write a PaneOrganizer class that instantiates the root node and returns it. Call public<prettify>start()</prettify> method. Instantiate a label and add it to the root node. Factor out code for timeline into its own method.
3. In setupTimeline(), instantiate a KeyFrame passing in a Duration and an instance of TimeHandler. Call method. Then instantiate a Timeline, passing in our KeyFrame, and play the Timeline.
4. Write a private Inner class that implements TimeHandler – it should know about Label and updates text on every ActionEvent.

Clock: PaneOrganizer

Within setupTimeline():
A. Instantiate a KeyFrame
B. Instantiate a Timeline, passing in our new KeyFrame
C. Set the CycleCount to INDEFINITE
D. Play, i.e. startTimeline
public class ExampleApp extends Application {
  import java.util.Date;
  import javafx.animation.Timeline;
  import javafx.animation.Animation;
  import javafx.event.*;
  //package includes Action Event,EventHandler classes
  import javafx.scene.control.*;
  //package includes Label,Button classes
  import javafx.scene.layout.*;
  //package includes Pane class and its sub classes
  import javafx.application.*;
  import javafx.scene.Scene;
  import javafx.stage.Stage;

  A. The last step is to create our TimeHandler and implement handle(ActionEvent), specifying what to occur at the end of each KeyFrame—called automatically by JFX
  B. Java.util.Date represents a specific instant in time. Data is a representation of the time, to the nearest millisecond, at the moment the Date is instantiated
  C. Because our Timeline has a Duration of 1 second, each second an Event will be generated, converted to a String, and set as the _labelText. This will appropriately update the Label with correct time every second!

  public class PaneOrganizer {
    private class TimeHandler implements EventHandler<ActionEvent> {
      public void handle(ActionEvent event) {
        //other code elided
      }
    }
    public void setupTimeline() {
      new Timeline(new KeyFrame(Duration.seconds(1),
        event -> _label.setText(now.toString())));
    }
    public void createGUI() {
      _root = new VBox();
      _label = new Label();
      _root.getChildren().add(_label);
      _root.setSpacing(1/2);
      LayoutPanes();
      _root.getChildren().add(buttonBox);
      Scene scene = new Scene(_root.getSceneRoot(), 200, 200);
    }
  }

  //end of PaneOrganizer class

  Clock: TimeHandler Private Inner Class

  A. The last step is to create our TimeHandler and implement handle(ActionEvent), specifying what to occur at the end of each KeyFrame—called automatically by JFX
  B. Java.util.Date represents a specific instant in time. Data is a representation of the time, to the nearest millisecond, at the moment the Date is instantiated
  C. Because our Timeline has a Duration of 1 second, each second an Event will be generated, converted to a String, and set as the _labelText. This will appropriately update the Label with correct time every second!

  public class PaneOrganizer {
    private class TimeHandler implements EventHandler<ActionEvent> {
      public void handle(ActionEvent event) {
        //other code elided
      }
    }
    public void setupTimeline() {
      new Timeline(new KeyFrame(Duration.seconds(1),
        event -> _label.setText(now.toString())));
    }
    public void createGUI() {
      _root = new VBox();
      _label = new Label();
      _root.getChildren().add(_label);
      _root.setSpacing(1/2);
      LayoutPanes();
      _root.getChildren().add(buttonBox);
      Scene scene = new Scene(_root.getSceneRoot(), 200, 200);
    }
  }

  //end of PaneOrganizer class

  Clock: TimeHandler Private Inner Class

  A. The last step is to create our TimeHandler and implement handle(ActionEvent), specifying what to occur at the end of each KeyFrame—called automatically by JFX
  B. Java.util.Date represents a specific instant in time. Data is a representation of the time, to the nearest millisecond, at the moment the Date is instantiated
  C. Because our Timeline has a Duration of 1 second, each second an Event will be generated, converted to a String, and set as the _labelText. This will appropriately update the Label with correct time every second!

  public class PaneOrganizer {
    private class TimeHandler implements EventHandler<ActionEvent> {
      public void handle(ActionEvent event) {
        //other code elided
      }
    }
    public void setupTimeline() {
      new Timeline(new KeyFrame(Duration.seconds(1),
        event -> _label.setText(now.toString())));
    }
    public void createGUI() {
      _root = new VBox();
      _label = new Label();
      _root.getChildren().add(_label);
      _root.setSpacing(1/2);
      LayoutPanes();
      _root.getChildren().add(buttonBox);
      Scene scene = new Scene(_root.getSceneRoot(), 200, 200);
    }
  }

  //end of PaneOrganizer class

  The Whole App: Clock

  public class PaneOrganizer {
    private class TimeHandler implements EventHandler<ActionEvent> {
      public void handle(ActionEvent event) {
        //other code elided
      }
    }
    public void setupTimeline() {
      new Timeline(new KeyFrame(Duration.seconds(1),
        event -> _label.setText(now.toString())));
    }
    public void createGUI() {
      _root = new VBox();
      _label = new Label();
      _root.getChildren().add(_label);
      _root.setSpacing(1/2);
      LayoutPanes();
      _root.getChildren().add(buttonBox);
      Scene scene = new Scene(_root.getSceneRoot(), 200, 200);
    }
  }

  //end of PaneOrganizer class

  Layout Panes

  • Until now, we have been adding all our GUI components to a VBox
  • VBoxes lay everything out in one vertical column
  • What if we want to make some more interesting GUIs?
  • Use different types of layout panes!
    o VBox is just one of many JavaFX panes—there are many more options
    o we will introduce a few, but check out our documentation or JavaDocs for a complete list

  HBox

  • Similar to VBox—but lays everything out in a horizontal row (hence the name)
  • Example:
    // code for setting the scene added
    _root.getChildren().add(_label);
    _root.getChildren().add(buttonBox);
  • Like VBox, we can set the amount of horizontal spacing between each child in the HBox using the setSpacing(double) method

  BorderPane (1/2)

  • BorderPane lays out children in top, left, bottom, right and center positions
  • To add things visually, use setLeft(Node), setCenter(Node), etc.
    o this includes an implicit call to getChildren(), add(…)
  • Use any type of Node—Panes (with their own children), Buttons, Labels, etc.
BorderPane (2/2)

- Remember our VBox example from earlier?
  ```java
  VBox buttonBox = new VBox();
  Button b1 = new Button("Top");
  Button b2 = new Button("Middle");
  Button b3 = new Button("Bottom");
  buttonBox.getChildren().addAll(b1, b2, b3);
  buttonBox.setSpacing(8);
  buttonBox.setAlignment(Pos.TOP_CENTER);
  ```

- We can make our VBox the center of this BorderPane
  ```java
  BorderPane container = new BorderPane();
  container.setCenter(buttonBox);
  ```

- No need to use all regions—could just use a few of them

- Unused regions are "compressed", e.g. could have a two-region (left/right) layout without a center

Absolute Positioning

- Until now, all layout panes we have seen have performed layout management for us
- What if we want to position our GUI components freely ourselves?

- Need to set component's location to exact pixel location on screen
  - called absolute positioning

- When would you use this?
  - to position shapes—stay tuned!

Pixels and Coordinate System

- Screen is a grid of pixels (tiny squares, each with RGB components)
- Cartesian plane with:
  - origin in upper-left corner
  - x-axis increasing left to right
  - y-axis increasing top to bottom
  - corresponds to English writing order

Pane

- Pane allows you to layout things completely freely—like on an art canvas

- It is a concrete superclass to all more specialized layout panes seen earlier that do automatic positioning
  - can call methods on its children (panes, buttons, shapes, etc.) to set location within pane
    - for example use setX(double) and setY(double) to position a Rectangle
  - Pane performs no layout management, so coordinates you set determine where things appear on the screen

Creating Custom Graphics

- We’ve now introduced you to using JavaFX's native UI elements
  - ex: Label and Button

- Lots of handy widgets for making your own graphical applications!

- What if you want to create your own custom graphics?
  - This lecture: build your own graphics using the javafx.scene.shape package!

JavaFX Scene Shape Package

- JavaFX provides built-in classes to represent 2D shapes, such as rectangles, ellipses, polygons, etc.

- All these classes inherit from abstract class Shape, which inherits from Node
  - methods relating to rotation and visibility are defined in Node
  - methods relating to color and border are defined in Shape
  - other methods are implemented in the individual classes of Ellipse, Rectangle, etc.
**Announcements**

- TA Safe House is due this weekend!
  - Early hand in: this Friday (10/7) at 10 PM
  - On time hand in: this Sunday (10/9) at 11:59 PM
  - There is no late deadline for this assignment!
- Cartoon out on Tuesday!
  - No design discussion + individual check in with discussion TA instead
  - TAs will reach out by email by end of day on Monday
- Review Session today at 7:30 pm-9 pm, MacMillan 115
- Have a great long weekend!

**Shape Constructors**
- Rectangle(double width, double height)
- Ellipse(double radiusX, double radiusY)
- Polygon(double... points)
  - The "..." in the signature means that you can pass in as many points as you would like to the constructor
  - Passing less than the number of x and y coordinates will result in a constructor failure
- Creating a new Polygon(x, y, x, y, x, y, x, y)

**Shapes: Setting Location**
- JavaFX Shapes have different behaviors (methods) for setting their location within their parent's coordinate system
  - Rectangle: use setX(double) and setY(double)
  - Ellipse: use setCenterX(double) and setCenterY(double)
  - Polygon: use setLocation(double) and setLocation(double)
- JavaFX has many different ways to set location
  - From our experience, this is the most straightforward way
  - If you choose to use other methods, be sure you fully understand them or you may get strange bugs!
  - Check out our JavaFX documentation and the javadocs for more detailed explanations!

**Shapes: Setting Size**
- JavaFX Shapes also have different behaviors (methods) for altering their size
  - Rectangle: use setWidth(double) and setHeight(double)
  - Ellipse: use setRadiusX(double) and setRadiusY(double)
  - Polygon: use setScaleX(double) and setScaleY(double)
  - Multiplying the original size in the X or Y dimension by the scale factor
- Again, this is not the only way to set size for Shapes, but it is relatively painless
  - Reminder: JavaFX documentation and Javadocs

**Accessors and Mutators of all Shapes**
- Rotation:
  - public final void setRotation(double rotation); public final double getRotation();
- Visibility:
  - public final void setVisible(boolean visible); public final boolean getVisible();
- Color:
  - public final void setFill(Paint color); public final Paint getFill();
  - public final void setStroke(Paint value); public final Paint getStroke();
  - public final void setStrokeWidth(double val); public final double getStrokeWidth();
- Border:
  - public final void setBorderWidth(double val); public final double getBorderWidth();

**JavaFX**
- Text: Use a color, which should never be white. Use predefined color constants: Color.BLACK, Color.BLUE, etc., or define your own new color by using the following syntax:
  - Paint color = Color.Hsb(100.0, 0.5, 0.3);