Lab 10: JavaFX

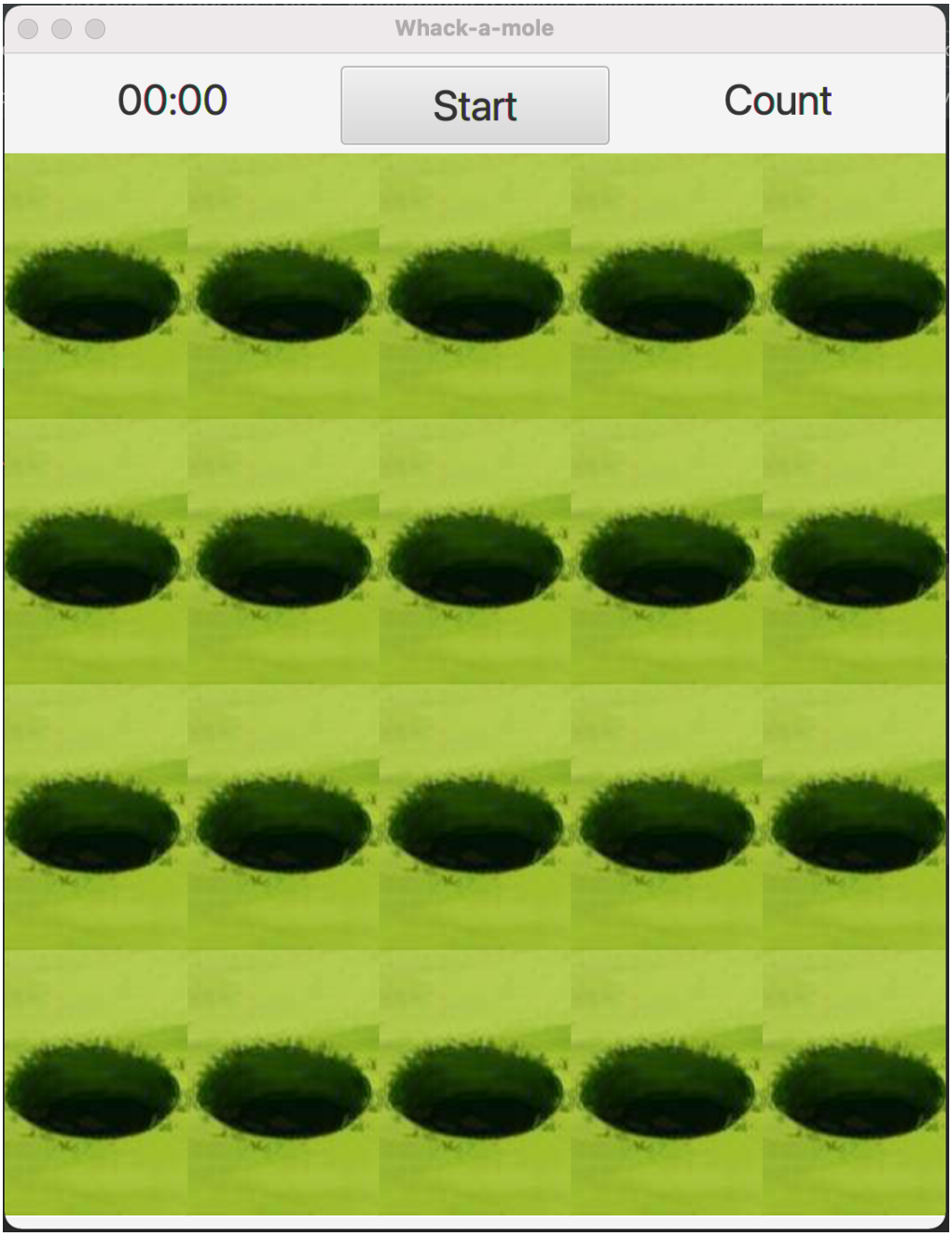
Author: Yida Tao, Yao Zhao

Whack-a-mole Game

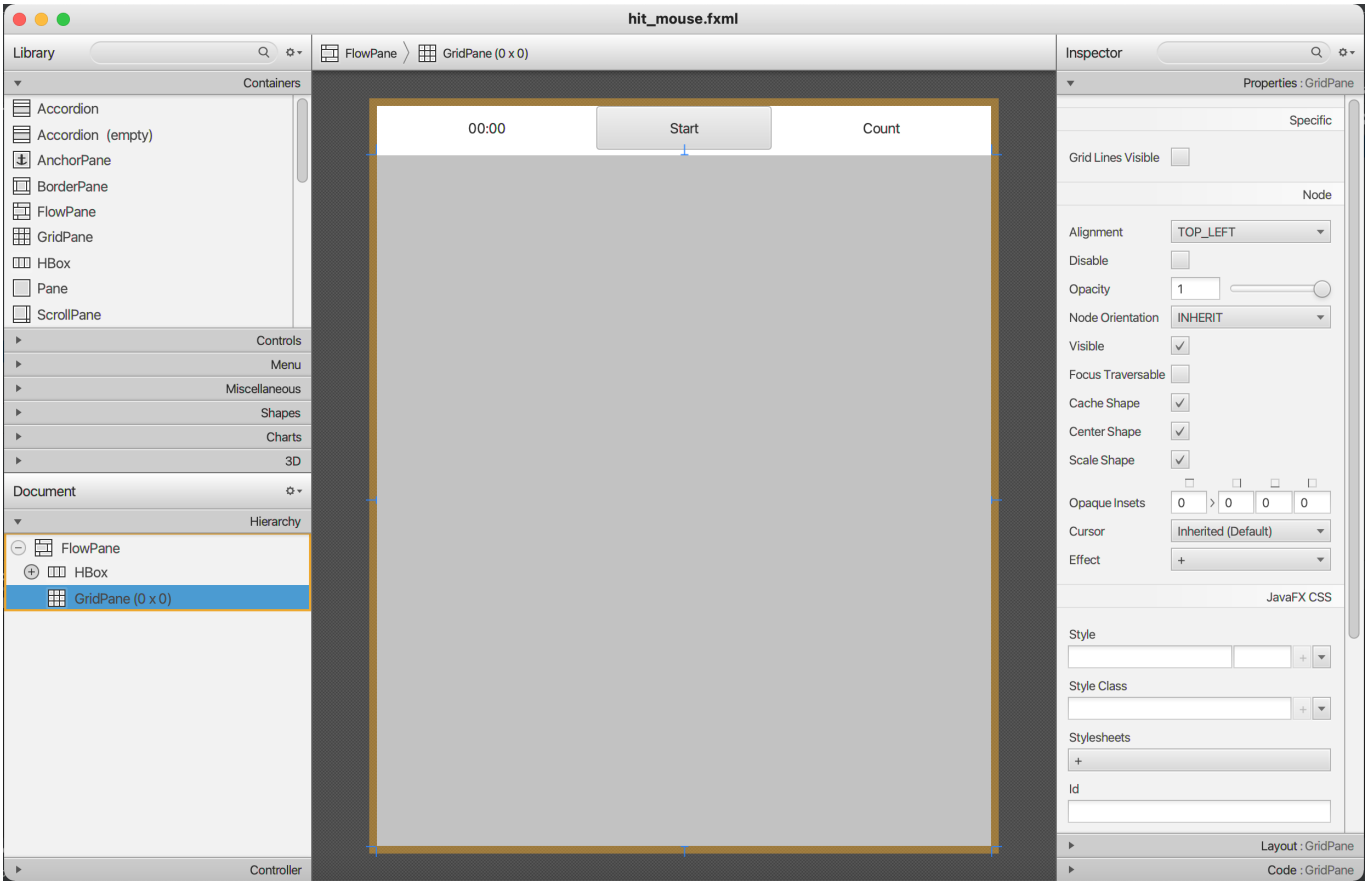
In this tutorial, we will develop a Whack-a-Mole game using JavaFX to demonstrate multithreading, event handling, and concurrency in a graphical user interface. This project will guide you through designing UI components with FXML, implementing asynchronous tasks, and efficiently handling user interactions.

1. Designing the Game UI

The initial game interface is shown below：



You can design the interface using **JavaFX Scene Builder**:



Corresponding FXML code:

```
<FlowPane fx:controller="com.example.whackamole.HitMouseController"
  xmlns:fx="http://javafx.com/fxml/1" alignment="center" hgap="5" vgap="5">
  <HBox fx:id="hbHead" prefWidth="560" prefHeight="40">
    <Label fx:id="labelTime" prefWidth="200" prefHeight="40" alignment="center"
text="00:00" />
    <Button fx:id="btnStart" prefWidth="160" prefHeight="40" alignment="center"
text="Start" />
    <Label fx:id="labelCount" prefWidth="200" prefHeight="40" alignment="center"
text="Count" />
  </HBox>
  <GridPane fx:id="gpGrass" prefWidth="560" prefHeight="630" />
</FlowPane>
```

Note: The `HitMouseController` class handles the game logic.

2. Defining Game States

Before implementing the game, consider the four possible states:

- the game starts with the holes empty (TYPE_HOLE)
- after a while, a mouse appear (TYPE_MOUSE)
- the player hits a mouse (TYPE_MOUSE_HIT)
- the player hits a hole (TYPE_HOLE_HIT)

These states are represented by the following images:

`hole.png`, `mouse.png`, `mouse_hit.png`, `hole_hit.png`

```
    private final static int TYPE_HOLE = 1; // hole
    private final static int TYPE_MOUSE = 2; // mouse
    private final static int TYPE_MOUSE_HIT = 3; // hit the mouse
    private final static int TYPE_HOLE_HIT = 4; // hit the hole
    private static Image imageHole; // hole image
    private static Image imageMouse; // mouse image
    private static Image imageMouseHit; // image of hit mouse
    private static Image imageHoleHit; // image of hit hole
    static {
        imageHole = new
Image(HitMouseController.class.getResourceAsStream("hole.png"));
        imageMouse = new
Image(HitMouseController.class.getResourceAsStream("mouse.png"));
        imageMouseHit = new
Image(HitMouseController.class.getResourceAsStream("mouse_hit.png"));
        imageHoleHit = new
Image(HitMouseController.class.getResourceAsStream("hole_hit.png"));
    }
```

3. Initializing Game Controls

Next, you need to initialize the controls on the UI, including the timer, reset the hit count, and the registration of the event for the start button, etc. You can add 4*5 buttons to the grid pane. Each button in the 4x5 grid represents a hole, and clicking a button indicates a hammer strike.

It is the code that initializes each control:

```
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        // Initialization after the UI is opened
        // initialize each hole button and set the click event for each hole
button
        for (int i = 0; i < btnArray.length; i++) {
            for (int j = 0; j < btnArray[i].length; j++) {
                btnArray[i][j] = getHoleView(); // get a Hole button
                Button view = btnArray[i][j];
                gpGrass.add(view, j, i + 1); // add the hole button to the grass
```

```
grid
                int x = i, y = j;
                // Sets the action event for the hole button.
                // Clicking a hole in the ground means swinging the hammer to
whack a mouse
                // default kick the hole
                view.setOnAction(e -> doAction(x, y, TYPE_HOLE_HIT));
            }
        }
        labelTime.setFont(Font.font("KaiTi", 25));
        btnStart.setFont(Font.font("KaiTi", 25));
        labelCount.setFont(Font.font("KaiTi", 25));
        btnStart.setOnAction(e -> {
            // event handler after clicking the start button
            isRunning = !isRunning;
            if (isRunning) { // if the game state is running
                btnStart.setText("Stop");
                hitCount = 0; // clear hitCount
                timeCount = 0; // clear timeCount
                beginTime = new Date().getTime(); // get the beginning time
                new MouseThread(0).start(); // start the first mouse thread
                new MouseThread(timeUnit * 1).start(); // start the second mouse
thread
                new MouseThread(timeUnit * 2).start(); // start the third mouse
thread
            } else { // game over
                btnStart.setText("Start");
            }
        });
    }
```

4. Multithreading and Concurrency

The game supports a maximum of three mice appearing simultaneously, so when you click the start button, only 3 mouse threads are started. Each `MouseThread` controls a single mouse:

(1) A mouse should emerge from a **randomly** chosen **empty** hole

(2) The mouse remains visible for a few seconds to allow the player has enough time to hit it.

```
    private class MouseThread extends Thread {

        private int mDelay; // Delay interval

        public MouseThread(int delay) {
            mDelay = delay;
        }

        public void run() {
            try {
                sleep(mDelay); // different mouse has different delay
```

```
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            while (isRunning) { // the game state is running
                int i = 0, j = 0;
                while (true) {
                    // Randomly generate the position where the mouse appears
                    i = new Random().nextInt(btnArray.length);
                    j = new Random().nextInt(btnArray[0].length);
                    if (timeArray[i][j] == 0) {
                        //do some action when the mouse go out the hole
                        doAction(i, j, TYPE_MOUSE);
                        break;
                    }
                }
                long nowTime = new Date().getTime();
                timeCount = (int) ((nowTime - beginTime) / 1000);
                try {
                    sleep((timeUnit - 100) * 3); // the time of the mouse stay out
the hole
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

5. Handling User Interactions and UI Updates

5.1 Handling game states

The `doAction` method handles the game states:

```
    // do some actions when the hole changes its state
    private synchronized void doAction(int i, int j, int type) {
        timeArray[i][j] = 3; // The mouse will stay out the hole for 3 seconds
        Button btn = btnArray[i][j];
        if (type == TYPE_HOLE_HIT) {
            showView(btn, imageHoleHit); // Show image of hit hole
            timeSchedule(i, j); // The hole timer began to count down
        } else if (type == TYPE_MOUSE) {
            showView(btn, imageMouse); // Show mouse image
            timeSchedule(i, j); // The hole timer began to count down
            btn.setOnAction(e -> { // Register the click event for the hole button
                doAction(i, j, TYPE_MOUSE_HIT); // Once the mouse in the hole is
hit, do the TYPE_MOUSE_HIT action
                hitCount++; // Update hitCount
            });
        } else if (type == TYPE_MOUSE_HIT) {
            showView(btn, imageMouseHit);  // Show image of hit mouse
            btn.setOnAction(null); // Unregister the click event for the hole
```

```
button
        }
    }
```

5.2.Updating the UI

The `showView` method uses `Task` to update the UI asynchronously.

The `Task`class is in `javafx.concurrent` package. See here for a further introduction.

```
    private void showView(Button btn, Image image) {
        // define a JavaFX Task
        // The call method of a task cannot manipulate the interface;
        // the succeeded method does
        Task task = new Task<Void>() {

            // The thread inside the call method is not the main thread
            // and cannot manipulate the interface
            protected Void call() throws Exception {
                return null;
            }

            // The thread inside the succeeded method is the main thread
            //can manipulate the interface
            protected void succeeded() {
                super.succeeded();
                btn.setGraphic(new ImageView(image)); // Set the button image as
the input image
                labelCount.setText(String.format("Hit %d mice", hitCount));
                labelTime.setText(String.format("%02d:%02d", timeCount / 60,
timeCount % 60));
            }
        };
        task.run(); // start the JavaFX task
    }
```

5.3.Resetting Holes

After a hole was hit, whether hits the mounse or not, it will be restored to empty hole after a while.

`TimerTask` is used here, and the detailed implementation code is as follows:

```
    private void timeSchedule(int i, int j) {
        Button btn = btnArray[i][j];
        Timer timer = new Timer();
        timer.schedule(new TimerTask() { // The timer is scheduled once per second
            public void run() {
                timeArray[i][j]--;
                if (timeArray[i][j] <= 0) { // time out
```

```
                    showView(btn, imageHole); // show empty hole
                    btn.setOnAction(e -> { // Registers the click event for the
hole
                        doAction(i, j, TYPE_HOLE_HIT);
                    });
                    timer.cancel(); // Cancel the timer
                }
            }
        }, 0, timeUnit);
    }
```

Reference:

- https://www.jetbrains.com/help/idea/javafx.html
- https://www.jetbrains.com/help/idea/opening-fxml-files-in-javafx-scene-builder.html#open-in-scene-builder
- https://jenkov.com/tutorials/javafx/concurrency.html
- https://github.com/aqi00/java/tree/master/chapter15/src/com/concurrent/mouse