CSC 360 - Software Development with AI

Exam II Notes

This is a closed book, closed note exam, that will have a format similar to the quizzes. The exam will consist primarily of multiple-choice questions with some short answer questions that are related to reading and understanding the execution of code, including unit testing using JUnit and developing and understanding JavaFX applications. You should also understand *when* to use specific coding constructs (e.g., when should you use a *ParameterizedTest*, when should you use a *BorderPane* layout?). Handouts of code will be provided for select questions.

The exam will cover the specific concepts that are outlined below.

Maven

1. Lifecycles (clean, default, and site) and associated steps/phases
2. Format of the POM file and tags described in the notes

JUnit and unit testing

1. AAA pattern for testing (Arrange-Act-Assert)
2. JUnit annotations: @Test, @BeforeEach, @AfterEach, @BeforeAll, @AfterAll
3. JUnit assertions: assertEquals, assertTrue, assertArrayEquals, assertThrows
4. Best practices for unit testing
5. JaCoCo metrics and interpreting JaCoCo and VS Code coverage reports
6. Calculating cyclomatic complexity
7. Parameterized tests and corresponding annotations: @ParameterizedTest, @ValueSource, @CsvSource, @NullSource, @EmptySource, @NullAndEmptySource
8. Test-driven development

JavaFX

1. JavaFX application lifecycle
2. Structure of a JavaFX application window: stage, scene, root node, and children nodes
3. Common User Interface Controls: TextField, TextArea, Button, ComboBox
4. Common Layouts: HBox, VBox, BorderPane, GridPane, Pane
5. Adding 2D shapes to the above layouts
6. Drawing on a *Canvas* using a *GraphicsContext* object
7. Understanding code involving *setOnAction*
8. Event handling: *addEventFilter* vs. *addEventHandler* and related events (e.g., *setOnKeyPressed*), *vs. ActionEvent* (*setOnAction*)
9. Using an AnimationTimer, including *handle*, *start*, and *stop* methods

Functional Interfaces and Lambda Expressions

1. Writing lambda expressions
2. Writing and calling methods with parameters that implement a functional interface

Additional Practice Questions

The best way to prepare for the exam is to review the previous quizzes, notes, and labs. Additional practice questions are provided below.

1. Consider the code below to test a method that takes a string and replaces a vowel with an asterisk. Add the appropriate annotation for a parameterized test that tests appropriate cases, including edge (except for the possibility of a *null* string)

```
void replacesVowelsAcrossVariousInputs(String input, String expected){
      assertEquals(expected, StringUtils.replaceVowels(input));
}
```

2. If the String is *null*, this should throw an invalid argument error. A test for this is below. Rewrite the test using a parameterized test.

```
@Test
void throwsIllegalArgumentExceptionForNullInput() {
   assertThrows(IllegalArgumentException.class, () -> StringUtils.replaceVowels(null));
}
```

3. Taking a Test-Driven Development approach, create a test for a method that computes the area of a rectangle for a given *length* and *width*, whose values must be positive.

4. For this JaCoco result, what non test methods have no coverage?

5. Consider this JaCoco result:

    a. What is the cyclomatic complexity of the *calculate* method
    b. Describe what additional test or tests are needed to get 100% coverage of the *calculate* method.

6. Write a lambda expression that takes an integer and returns the larger of the integer or 0.
7. The *Consumer* functional interface is described here.
    a. Write a method called *validate* that takes a *String* input and an object that implements the *Consumer<String>* Functional interface. If the string is not *null*, then *validate* returns *true;* otherwise, the method returns *false* and outputs *[X] input cannot be null*, where *[X]* is defined by the functional interface (see part (b)).
    b. Call the *validate* method so that if the input is not valid, the output will be *Error: input cannot be null*.
8. The *start* method of a JavaFX application is shown below. What does each line do? What is displayed on the screen when the application runs? Note that the definition of *strokeLine* is given below.

```
void javafx.scene.canvas.GraphicsContext.strokeLine(double x1, double y1, double x2, double y2)
```

Strokes a line using the current stroke paint.

This method will be affected by any of the global common or stroke attributes as specified in the Rendering Attributes Table.

- **Parameters:**
    - **x1** the X coordinate of the starting point of the line.
    - **y1** the Y coordinate of the starting point of the line.
    - **x2** the X coordinate of the ending point of the line.
    - **y2** the Y coordinate of the ending point of the line.

```
public void start(Stage stage) {
    double width = 400;
    double height = 400;

    Canvas canvas = new Canvas(width, height);
    GraphicsContext gc = canvas.getGraphicsContext2D();

    gc.setFill(Color.WHITE);
    gc.fillRect(0, 0, width, height);

        gc.setStroke(Color.RED);
    gc.setLineWidth(4);
```

```
        gc.strokeLine(width / 2, 0, width / 2, height);
        gc.strokeLine(0, height / 2, width, height / 2);

        stage.setScene(new Scene(new StackPane(canvas)));
        stage.setTitle("Practice Question");
        stage.show();
    }
```

9. Look at the code for the Exam II Practice Code. Describe what each line of code does. Describe what the application looks like. Overall, what does the code do? Note: you can run the code to help understand it, but you should be able to answer these questions without running the code.