CMSC 215

Programming Project 4 – Time Interval Checker

Daniel Smolsky

Section 1 – Approach

To address this programming project, a systematic approach is adopted, focusing on developing four key classes: Interval, Time, InvalidTime, and Project4. The Interval class is designed as a generic, immutable type to manage time intervals, requiring precise implementation of methods to evaluate interval relationships. The Time class encapsulates hours, minutes, and meridian, implementing Comparable for time comparison and ensuring rigorous input validation to prevent invalid time creation. The InvalidTime exception class facilitates error signaling, particularly for time validation errors. The Project4 class integrates these components into a JavaFX GUI, enabling user interaction for interval comparison and time checking

Section 2 – Assumptions

In solving the project, the program makes several assumptions that are critical to its design and functionality. Firstly, it assumes that all time inputs adhere to a 12-hour format with AM/PM notation, requiring users to input times in this specific format for the Time class to process them correctly. This assumption extends to the expectation that the string representation of time is correctly formatted before parsing. Secondly, the program assumes that the Comparable interface's implementation by the Time class provides a sufficient basis for comparing time objects, which is essential for the Interval class's methods to function correctly. It also presumes that users will input valid time intervals, where the start time precedes the end time, as it may not explicitly validate this relationship unless implemented within the Interval constructor or methods. Additionally, the program assumes that the GUI interface provided by JavaFX will be an effective medium for user interaction, relying on users to correctly interpret and respond to the GUI prompts. Lastly, it assumes that exceptions, particularly InvalidTime, will be adequately handled to inform users of input errors without causing the program to terminate unexpectedly.

Section 3 – Not Implemented

Section 4 – User Guide

To use the Time Interval Checker program, start by launching the application, which will present you with a user-friendly interface. Enter the start and end times for two intervals in the format HH:MM AM/PM into the designated text fields. To compare these intervals, click the "Compare Intervals" button; the program will then display whether the intervals overlap, are disjoint, or if one is a sub-interval of the other. For checking if a specific time falls within these intervals, enter the time in the same format into the "Time to Check" field and click the "Check Time" button. The result will indicate whether the time is within neither, either, or both intervals. The program ensures ease of use with clear instructions and immediate feedback displayed on the interface.

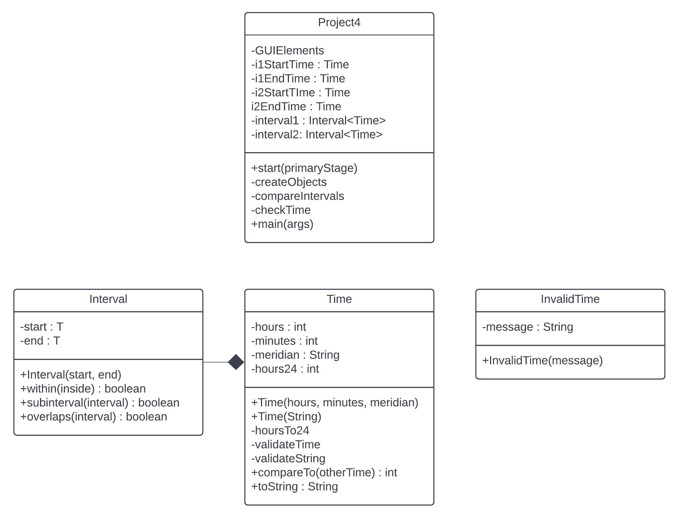
Section 5 – Lessons Learned

Implementing this project taught valuable lessons in handling date and time manipulation in Java, emphasizing the importance of understanding and correctly applying the Comparable interface for custom objects. It also underscored the significance of robust input validation and exception handling to ensure user inputs are within expected parameters, preventing runtime errors. Additionally, integrating a graphical user interface (GUI) with JavaFX highlighted the intricacies of building user-friendly applications, from layout design to event handling, showcasing the practical application of object-oriented programming principles and the necessity of clear, user-centric design and feedback mechanisms.

Section 6 – Possible Improvements

Possible improvements could include adding input validation directly within the GUI to provide immediate feedback to users before attempting to process the intervals, reducing the chance of encountering InvalidTime exceptions due to user input errors. Another enhancement could be the introduction of a clear or reset button to quickly clear all input fields and results, improving the user experience by allowing for easier multiple uses of the application without manually clearing each field. Additionally, implementing a more dynamic layout or responsive design could improve usability across different screen sizes and resolutions.

Section 7 – UML



Section 8 – Source Code

*/\*\**

*\* Daniel Smolsky*

*\* Programming Project 4 - Time Interval Checker*

*\* February 28, 2017*

*\* Represents a generic interval between two points of a comparable type.*

*\* This class provides methods to check if a value or another interval is within this interval,*

*\* if it overlaps with another interval, or if another interval is a subinterval of this one.*

*\**

*\* @param <T> the type of the interval boundaries, which must be Comparable*

*\*/*

package Project4;

public class Interval <T extends Comparable<T>> {

    private T start;

    private T end;

    public Interval(T *start*, T *end*) {

        this.start = *start*;

        this.end = *end*;

    }

*// returns true if the parameter is within the interval*

    public boolean **within**(T *inside*) {

        return start.compareTo(*inside*) >= 0 && end.compareTo(*inside*) <= 0;

    }

*// returns true if paremeter is a subinterval of this instance*

    public boolean **subinterval**(Interval<T> *interval*) {

        return this.within(*interval*.start) && this.within(*interval*.end);

    }

*// returns true if the intervals overlap*

    public boolean **overlaps**(Interval<T> *interval*) {

        return (!this.within(*interval*.start) && this.within(*interval*.end)) ||

        (this.within(*interval*.start) && !this.within(*interval*.end));

    }

}

*/\*\**

*\* Daniel Smolsky*

*\* Programming Project 4 - Time Interval Checker*

*\* February 28, 2017*

*\* Represents a time in either 12-hour or 24-hour format.*

*\* This class provides methods to set and validate time in 12-hour format with AM/PM notation,*

*\* and internally converts and stores the time in 24-hour format for comparison purposes.*

*\* It implements the Comparable interface to allow comparison between Time objects.*

*\*/*

package Project4;

public class Time implements Comparable<Time> {

    private int hours;

    private int minutes;

    private String meridian;

    private int hours24; *// this variable will store hours converted to military format (24-hour clock)*

    public Time (int *hours*, int *minutes*, String *meridian*) throws InvalidTime {

        this.hours = *hours*;

        this.minutes = *minutes*;

        this.meridian = *meridian*;

        validateTime();

        hoursTo24();

    }

    public Time (String *timeString*) throws InvalidTime {

        validateString(*timeString*);

        this.hours = Integer.parseInt(*timeString*.substring(0, 2));

        this.minutes = Integer.parseInt(*timeString*.substring(3, 5));

        this.meridian = *timeString*.substring(6, 8);

        validateTime();

        hoursTo24();

    }

*// calculate 24-hour format*

    private void **hoursTo24**() {

        if ("PM".equals(meridian) && hours != 12)

            hours24 = hours + 12;

        else if ("AM".equals(meridian) && hours == 12)

            hours24 = 0;

        else

            hours24 = hours;

    }

*// check input parameters match HH:MM AM*

    private void **validateTime** () throws InvalidTime {

        if (hours < 0 || hours > 12) {

            throw new InvalidTime("Hours must be between 00 and 12");

        }

        if (minutes < 0 || minutes > 59) {

            throw new InvalidTime("Minutes must be between 00 and 59");

        }

        if (!meridian.equals("AM") && !meridian.equals("PM")) {

            throw new InvalidTime("Must specify AM or PM");

        }

    }

*// check input string matches "HH:MM AM"*

    private void **validateString**(String *inString*) throws InvalidTime {

        if (*inString*.length() != 8) {

            throw new InvalidTime("Input string incorrect length");

        }

        try {

            Integer.parseInt(*inString*.substring(0, 2));

        } catch (NumberFormatException *e*) {

            throw new InvalidTime("Hours aren't numerical");

        }

        try {

            Integer.parseInt(*inString*.substring(3, 5));

        } catch (NumberFormatException *e*) {

            throw new InvalidTime("Minutes aren't numerical");

        }

    }

*// returns 1 if parameter is greater than instance, -1 if less, 0 if equal*

    @Override

    public int **compareTo** (Time *otherTime*) {

        if (*otherTime*.hours24 > this.hours24) {

            return 1;

        }

        else if (*otherTime*.hours24 < this.hours24) {

            return -1;

        }

*// if hours are equal, compare minutes*

        else {

            if (*otherTime*.minutes > this.minutes) {

                return 1;

            }

            else if (*otherTime*.minutes < this.minutes) {

                return -1;

            }

            else return 0; *//times are equal*

        }

    }

*// return string representation of the time*

    @Override

    public String **toString**() {

        return (hours < 10 ? "0" + hours : hours) + ":" + (minutes < 10 ? "0" + minutes : minutes) + " " + meridian;

    }

}

*/\*\**

*\* Daniel Smolsky*

*\* Programming Project 4 - Time Interval Checker*

*\* February 28, 2017*

*\* Represents an exception for invalid time inputs in the Time Interval Checker application.*

*\* This class extends the Exception class and is used to indicate that an invalid time has been provided.*

*\* It is thrown by the Time class when the time does not meet the specified format or value constraints.*

*\*/*

package Project4;

public class InvalidTime extends Exception {

    private String message;

    public InvalidTime(String *message*) {

        super(*message*);

        this.message = *message*;

    }

}

*/\*\**

*\* Daniel Smolsky*

*\* programming project 4 - Time Interval Checker*

*\* February 28, 2017*

*\* The Project4 class extends the Application class from JavaFX to create a GUI application.*

*\* This application allows users to input time intervals and a specific time to check against these intervals.*

*\* It provides functionality to compare two time intervals to see if they overlap and to check if a specific time falls within these intervals.*

*\* The GUI is built using JavaFX components such as Labels, TextFields, and Buttons, arranged in a GridPane layout.*

*\* Users can input times in a 12-hour format with AM/PM notation, and the application handles the conversion and comparison internally.*

*\*/*

package Project4;

import javafx.application.Application;

import javafx.scene.control.TextField;

import javafx.scene.control.Label;

import javafx.scene.control.Button;

import javafx.scene.layout.GridPane;

import javafx.scene.Scene;

import javafx.stage.Stage;

import javafx.geometry.Pos;

import javafx.geometry.HPos;

import javafx.geometry.Insets;

import javafx.scene.Node;

public class Project4 extends Application {

*// Create text fields and buttons and initizalize some values*

    private Label lbStartTime = new Label("Start Time");

    private Label lbEndTime = new Label("End Time");

    private Label lbTimeInterval1 = new Label("Time Interval 1");

    private TextField tfStartTimeI1 = new TextField("10:30 AM");

    private TextField tfEndTimeI1 = new TextField("12:30 PM");

    private Label lbTimeInterval2 = new Label("Time Interval 2");

    private TextField tfStartTimeI2 = new TextField("11:05 AM");

    private TextField tfEndTimeI2 = new TextField("01:00 PM");

    private Button btCompareIntervals = new Button("Compare Intervals");

    private Label lbTimeToCheck = new Label("Time to Check");

    private TextField tfTimeToCheck = new TextField();

    private Button btCheckTime = new Button("Check Time");

    private TextField tfResult = new TextField();

*// Create time and interval objects*

    private Time i1StartTime;

    private Time i1EndTime;

    private Time i2StartTime;

    private Time i2EndTime;

    private Time timeToCheck;

    private Interval<Time> interval1;

    private Interval<Time> interval2;

    public void **start**(Stage *primaryStage*) {

*// Create UI*

        GridPane gridPane = new GridPane();

        gridPane.setHgap(5);

        gridPane.setVgap(5);

        gridPane.setPadding(new Insets(10, 10, 10, 10));

*// Place nodes*

        gridPane.add(lbStartTime, 1, 0);

        gridPane.add(lbEndTime, 2, 0);

        gridPane.add(lbTimeInterval1, 0, 1);

        gridPane.add(tfStartTimeI1, 1, 1);

        gridPane.add(tfEndTimeI1, 2, 1);

        gridPane.add(lbTimeInterval2, 0, 2);

        gridPane.add(tfStartTimeI2, 1, 2);

        gridPane.add(tfEndTimeI2, 2, 2);

        gridPane.add(btCompareIntervals, 0, 3);

        gridPane.add(lbTimeToCheck, 0, 4);

        gridPane.add(tfTimeToCheck, 1, 4);

        gridPane.add(btCheckTime, 0, 5);

        gridPane.add(tfResult, 0, 6);

*// Set properties for the UI*

        gridPane.setAlignment(Pos.CENTER);

        GridPane.setHalignment(lbStartTime, HPos.CENTER);

        GridPane.setHalignment(lbEndTime, HPos.CENTER);

        GridPane.setColumnSpan(btCompareIntervals, 3);

        btCompareIntervals.setPrefWidth(400);

        GridPane.setColumnSpan(tfTimeToCheck, 2);

        tfTimeToCheck.setPrefWidth(400);

        GridPane.setColumnSpan(btCheckTime, 3);

        btCheckTime.setPrefWidth(400);

        GridPane.setColumnSpan(tfResult, 3);

        tfResult.setPrefWidth(400);

        tfResult.setEditable(false);

        for (Node node : gridPane.getChildren()) {

            if (node instanceof TextField) {

                ((TextField)node).setAlignment(Pos.CENTER);

            }

        }

*// process events*

        btCompareIntervals.setOnAction(*e* -> {

            try {

                compareIntervals();

            } catch (InvalidTime *ex*) {

                tfResult.setText(ex.getMessage());

            }

        });

        btCheckTime.setOnAction(*e* -> {;

            try {

                checkTime();

            } catch (InvalidTime *ex*) {

                tfResult.setText(ex.getMessage());

            }

        });

*// create a scene and place it in the stage*

        Scene scene = new Scene(gridPane, 410, 210);

        gridPane.requestFocus();

*primaryStage*.setTitle("Time Interval Checker");

*primaryStage*.setScene(scene);

*primaryStage*.show();

    }

*// create time and interval objects*

    private void **createObjects**() throws InvalidTime {

        i1StartTime = new Time(tfStartTimeI1.getText());

        i1EndTime = new Time(tfEndTimeI1.getText());

        i2StartTime = new Time(tfStartTimeI2.getText());

        i2EndTime = new Time(tfEndTimeI2.getText());

        interval1 = new Interval<>(i1StartTime, i1EndTime);

        interval2 = new Interval<>(i2StartTime, i2EndTime);

    }

*// compare intervals and display result*

    private void **compareIntervals**() throws InvalidTime {

        createObjects();

        if (interval2.subinterval(interval1)) {

            tfResult.setText("Interval 1 is a sub-interval of interval 2");

        }

        else if (interval1.subinterval(interval2)) {

            tfResult.setText("Interval 2 is a sub-interval of interval 1");

        }

        else if (interval1.overlaps(interval2)) {

            tfResult.setText("The intervals overlap");

        }

        else {

            tfResult.setText("the intervals are disjoint");

        }

    }

*// check if time is within intervals and display result*

    private void **checkTime**() throws InvalidTime {

        createObjects();

        timeToCheck = new Time(tfTimeToCheck.getText());

        if (interval1.within(timeToCheck) && interval2.within(timeToCheck)) {

            tfResult.setText("Both intervals contain the time " + timeToCheck);

        }

        else if (interval1.within(timeToCheck)) {

            tfResult.setText("Only interval 1 contains the time " + timeToCheck);

        }

        else if (interval2.within(timeToCheck)) {

            tfResult.setText("Only interval 2 contains the time " + timeToCheck);

        }

        else {

            tfResult.setText("Neither interval contains the time " + timeToCheck);

        }

    }

    public static void **main**(String[] *args*) throws Exception {

        launch(*args*);

    }

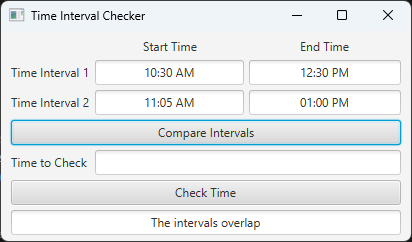
}

Section 9 – Test Plans

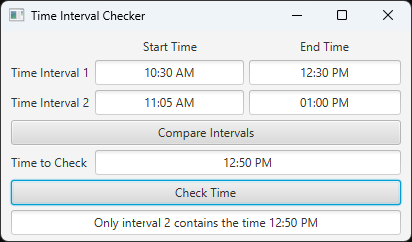
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test # | Purpose | Pos/Neg Test | Input Values | Expected Result | Pass/Fail |
| 1 | Confirm my program returns the same results as the example in assignment outline | Positive | 10:30 AM, 12:30 PM, 11:05 AM, 01:00 PM | The intervals overlap | Pass |
| 2 | Confirm my program returns the same results as the example in assignment outline | Positive | 10:30 AM, 12:30 PM, 11:05 AM, 01:00 PM 12:50 PM | Only interval 2 contains the time 12:50 PM | Pass |
| 3 | Confirm program outputs an error message when an incorrect input is detected | Positive | 10:30 AM, 12:30 PM, 11:05 AM, 01:00 PM 12:50 | Input string incorrect length | Pass |
| 4 | Confirm program outputs an error message when an incorrect input is detected | Positive | 14:30 AM, 12:30 PM, 11:05 AM, 01:00 PM  12:50 PM | Hours must be between 00 and 12 | Pass |
| 5 | Program fails for an interval where the start is later than end time | Negative | 08:00 AM, 04:00 AM  05:00 AM 06:00 AM | Interval 2 is a sub-interval of Interval 1 | Fail |
| 6 | Nodes are not bound to pane and resizing window leads to non-functional GUI | Negative | Modify the size of the scene | Some textfields or labels become unreadable | Pass |

Section 10 – Screen shots

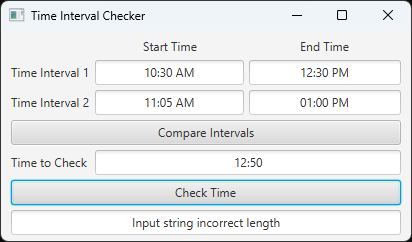
Test 1 –



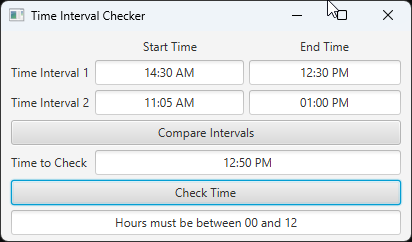
Test 2 –



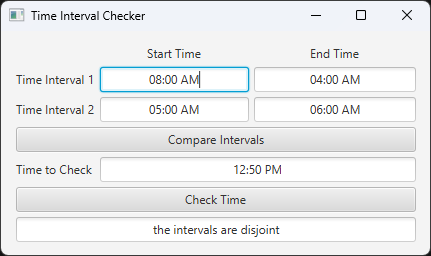
Test 3 –



Test 4 –



Test 5 –



Test 6 –

