Franklin Marathon Test-Driven Development Plan

TDD Requirement 4.1:

The system shall allow a runner to sign up for two races (a Saturday race and a Sunday race) and add a 20% discount to sum of the two races. When this happens, the runner's race roster information will be added to both rosters.

Verify no Regression
Use Registration from 2.1

Input	DOB: 19601030
	Registration: 20251031
	5k
Expected Output	25
Actual Output	25
Pass/Fail	Pass

Verify Bundling W/O Senior Discount
Use Registration from 2.1

Input	DOB: 19601030
	Registration: 20251029
	5k & half
Expected Output	76
Actual Output	76
Pass/Fail	Pass

Verify Bundling with Senior Discount
Use Registrant from 2.1

Input	DOB: 19601030
	Registration: 20251031
	5k & half
Expected Output	66
Actual Output	66
Pass/Fail	Pass

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.Scanner;

public class req41 {
    private static final DateTimeFormatter DOB_FMT =
DateTimeFormatter.ofPattern("yyyyMMdd");
    private static final DateTimeFormatter REG_TS_FMT =
DateTimeFormatter.ofPattern("yyyyMMddHHmmss");
    private static final String _5K_ROSTER_FILE = "5k_RaceRoster";
    private static final String _10K_ROSTER_FILE = "10k_RaceRoster";
    private static final String HALF_ROSTER_FILE = "Half_RaceRoster";
    private static final String FULL_ROSTER_FILE = "Full_RaceRoster";

    // Which field indices (0-based) should be compared to determine duplicates.
    // Default: compare firstName(0), lastName(1), gender(3), email(4), regTs(5)
    private static final int[] FIELDS_TO_COMPARE = {0, 1, 3, 4, 5};

    private static final String DELIM = "|";
    private static final String DELIM_REGEX = "\\|";

    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter first name :");
        String firstName = sc.next().trim();
        System.out.print("Enter last name :");
        String lastName = sc.next().trim();
        System.out.print("Enter date of birth (YYYYMMDD) :");
        String dobStr = sc.next().trim();
        System.out.print("Enter gender :");
        String gender = sc.next().trim();
        System.out.print("Enter email :");
        String email = sc.next().trim();
        System.out.print("Enter registration timestamp (YYYYMMDD or YYYYMMDDHHMMSS)
:");
        String regTs = sc.next().trim();
        System.out.print("Enter Saturday race category (5k or 10k) :");
        String satRace = sc.next().trim().toLowerCase();
        System.out.print("Enter Sunday race category (half or full) :");
        String sunRace = sc.next().trim().toLowerCase();
        sc.close();

        LocalDate dob = parseToLocalDate(dobStr);
        LocalDate regDate = parseToLocalDate(regTs);
        String SATURDAY_ROSTER = setRoster(satRace);
        String SUNDAY_ROSTER = setRoster(sunRace);

        int raceYear = regDate.getYear() + (regDate.getMonthValue() >= 6 ? 1 : 0);
        LocalDate tday = req11.computeTDay(raceYear);
        LocalDate satRaceDate = tday.plusDays(2);
        LocalDate sunRaceDate = tday.plusDays(3);

        int satRaceAge = req12.calculateAge(dob, satRaceDate);
        int sunRaceAge = req12.calculateAge(dob, sunRaceDate);
        int ageOnRegister = req12.calculateAge(dob, regDate);
        int satCost = !satRace.isEmpty() ? req31.determineFee(satRace,
req11.determineRacePeriod(regDate)) : 0;
        int sunCost = !sunRace.isEmpty() ? req31.determineFee(sunRace,
req11.determineRacePeriod(regDate)) : 0;
        double cost = (satCost != 0 && sunCost != 0) ? (satCost + sunCost) * 0.8 -
(ageOnRegister > 64 ? 10 : 0)
                                                     : satCost + sunCost -
(ageOnRegister > 64 ? 5 : 0);

        // Build single-line, delimited entries to make records resilient to file
corruption
        String satEntry = !satRace.isEmpty() ? String.join(DELIM,
            firstName,
            lastName,
            Integer.toString(satRaceAge),
            gender,
            email,
            regTs,
            Double.toString(cost)) : null;

        String sunEntry = !sunRace.isEmpty() ? String.join(DELIM,
            firstName,
            lastName,
            Integer.toString(sunRaceAge),
            gender,
            email,
            regTs,
            Double.toString(cost)) : null;

        if (satEntry != null && !isDuplicate(satEntry, true)) {
            writeRecord(SATURDAY_ROSTER, satEntry);
            System.out.println("Adding entry to " + SATURDAY_ROSTER);
        } else {
            System.out.println("Already registered for a saturday race.");
        }
        if (sunEntry != null && !isDuplicate(sunEntry, false)) {
            writeRecord(SUNDAY_ROSTER, sunEntry);
            System.out.println("Adding entry to " + SUNDAY_ROSTER);
        } else {
            System.out.println("Already registered for a sunday race.");
        }
    }

    public static String setRoster(String raceCat) {
        switch (raceCat) {
            case "5k":
                return _5K_ROSTER_FILE;
            case "10k":
                return _10K_ROSTER_FILE;
            case "half":
                return HALF_ROSTER_FILE;
            case "full":
                return FULL_ROSTER_FILE;
            default:
                return null;
        }
    }

    /**
     * Parse an input string that may be either a date (yyyyMMdd) or a
     * timestamp (yyyyMMddHHmmss) and return the corresponding LocalDate.
     * If parsing both patterns fails but the input contains at least 8 digits,
     * the first 8 digits are used as the date portion.
     *
     * Throws DateTimeParseException if no valid date can be extracted.
     */
    public static LocalDate parseToLocalDate(String input) {
        String s = input == null ? "" : input.trim();
        try {
            return LocalDate.parse(s, DOB_FMT);
        } catch (DateTimeParseException e) {
            // Fall through and try timestamp
        }

        // Try yyyyMMddHHmmss as LocalDateTime then convert
        try {
            LocalDateTime dt = LocalDateTime.parse(s, REG_TS_FMT);
            return dt.toLocalDate();
        } catch (DateTimeParseException e) {
            // Fall through to heuristic
        }

        String digits = s.replaceAll("\\D", "");
        if (digits.length() >= 8) {
            String datePart = digits.substring(0, 8);
            return LocalDate.parse(datePart, DOB_FMT);
        }

        throw new DateTimeParseException("Unparseable date: " + input, input, 0);
    }

    public static int writeRecord(String rosterFile, String entry) throws IOException {
        try (FileWriter fw = new FileWriter(rosterFile, true);
             PrintWriter pw = new PrintWriter(fw)) {
            pw.println(entry);
        }
        return 0;
    }

    /*
     * Newest version of isDuplicate
     */
    private static boolean isDuplicate(String entry, boolean sat) throws IOException {
        String[] entryFields = entry.split(DELIM_REGEX, -1);

        // Build the target values using the configured indices.
        String[] target = new String[FIELDS_TO_COMPARE.length];
        for (int i = 0; i < FIELDS_TO_COMPARE.length; i++) {
            target[i] = entryFields[FIELDS_TO_COMPARE[i]].trim();
        }

        File[] files;
        if (sat) {
            files = new File[]{new File(_5K_ROSTER_FILE), new File(_10K_ROSTER_FILE)};
        } else {
            files = new File[]{new File(HALF_ROSTER_FILE), new File(FULL_ROSTER_FILE)};
        }

        // Process each roster file individually.
        for (File file : files) {
            if (!file.exists()) {   continue; }

            try (BufferedReader br = new BufferedReader(new FileReader(file))) {
                String line;

                while ((line = br.readLine()) != null) {
                    String[] recordFields = line.split(DELIM_REGEX, -1);
                    boolean matches = true;
                    for (int i = 0; i < FIELDS_TO_COMPARE.length; i++) {
                        int fieldIndex = FIELDS_TO_COMPARE[i];
                        if (fieldIndex >= recordFields.length) {
                            matches = false;
                            break;
                        }
                        if (!recordFields[fieldIndex].trim().equals(target[i])) {
                            matches = false;
                            break;
                        }
                    }
                    if (matches) { return true; }
                }
            }
        }
        return false;
    }
}
```

TDD Requirement 4.2:

The system will allow no more than 100 runners to sign up for events on Saturday, and the system will allow no more than 100 runners to sign up for events on Sunday.

Verify adding when @ 99 total	
Use Registration from 2.1	
Precondition	99 Saturday runners
Input	5k
Expected Output	Runner added to roster.
Actual Output	Runner added to roster.
Pass/Fail	Pass

Verify denial when @100	
Use Registration from 2.1	
Precondition	100 Sunday runners
Input	Half
Expected Output	Sunday rosters full.
Actual Output	Sunday rosters full.
Pass/Fail	Pass

Verify denial when @100	
Use Registration from 2.1	
Precondition	100 Saturday runners
Input	5k
Expected Output	Saturday rosters full.
Actual Output	Saturday rosters full.
Pass/Fail	Pass

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.Scanner;

public class req42 {
    private static final DateTimeFormatter DOB_FMT =
DateTimeFormatter.ofPattern("yyyyMMdd");
    private static final DateTimeFormatter REG_TS_FMT =
DateTimeFormatter.ofPattern("yyyyMMddHHmmss");
    private static final String _5K_ROSTER_FILE = "5k_RaceRoster";
    private static final String _10K_ROSTER_FILE = "10k_RaceRoster";
    private static final String HALF_ROSTER_FILE = "Half_RaceRoster";
    private static final String FULL_ROSTER_FILE = "Full_RaceRoster";

    // Simple per-file cached counts initialized at class load.
    public static int fiveK_count;
    public static int tenK_count;
    public static int half_count;
    public static int full_count;

    // Which field indices (0-based) should be compared to determine duplicates.
    // Default: compare firstName(1), lastName(2), gender(4), and email(5)
    private static final int[] FIELDS_TO_COMPARE = {1, 2, 4, 5};

    private static final String DELIM = "|";
    private static final String DELIM_REGEX = "\\|";

    public static void main(String[] args) throws IOException {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter first name :");
        String firstName = sc.next().trim();
        System.out.print("Enter last name :");
        String lastName = sc.next().trim();
        System.out.print("Enter date of birth (YYYYMMDD) :");
        String dobStr = sc.next().trim();
        System.out.print("Enter gender :");
        String gender = sc.next().trim();
        System.out.print("Enter email :");
        String email = sc.next().trim();
        System.out.print("Enter registration timestamp (YYYYMMDD or YYYYMMDDHHMMSS)
:");
        String regTs = sc.next().trim();
        System.out.print("Enter Saturday race category (5k or 10k) :");
        String satRace = sc.next().trim().toLowerCase();
        System.out.print("Enter Sunday race category (half or full) :");
        String sunRace = sc.next().trim().toLowerCase();
        sc.close();

        LocalDate dob = parseToLocalDate(dobStr);
        LocalDate regDate = parseToLocalDate(regTs);
        String SATURDAY_ROSTER = setRoster(satRace);
        String SUNDAY_ROSTER = setRoster(sunRace);

        int raceYear = regDate.getYear() + (regDate.getMonthValue() >= 6 ? 1 : 0);
        LocalDate tday = req11.computeTDay(raceYear);
        LocalDate satRaceDate = tday.plusDays(2);
        LocalDate sunRaceDate = tday.plusDays(3);

        int satRaceAge = req12.calculateAge(dob, satRaceDate);
        int sunRaceAge = req12.calculateAge(dob, sunRaceDate);
        int ageOnRegister = req12.calculateAge(dob, regDate);
        int satCost = !satRace.isEmpty() ? req31.determineFee(satRace,
req11.determineRacePeriod(regDate)) : 0;
        int sunCost = !sunRace.isEmpty() ? req31.determineFee(sunRace,
req11.determineRacePeriod(regDate)) : 0;
        double cost = (satCost != 0 && sunCost != 0) ? (satCost + sunCost) * 0.8 -
(ageOnRegister > 64 ? 10 : 0)
                                                    : satCost + sunCost -
(ageOnRegister > 64 ? 5 : 0);

        int satSeq = satRace.equals("5k") ? fiveK_count + 1 : tenK_count + 1;
        int sunSeq = sunRace.equals("half") ? half_count + 1 : full_count + 1;

        // Build single-line, delimited entries to make records resilient to file
corruption
        String satEntry = !satRace.isEmpty() ? String.join(DELIM,
            Integer.toString(satSeq),
            firstName,
            lastName,
            Integer.toString(satRaceAge),
            gender,
            email,
            regTs,
            Double.toString(cost)) : null;

        String sunEntry = !sunRace.isEmpty() ? String.join(DELIM,
            Integer.toString(sunSeq),
            firstName,
            lastName,
            Integer.toString(sunRaceAge),
            gender,
            email,
            regTs,
            Double.toString(cost)) : null;

        if (satEntry != null && !isDuplicate(satEntry, true)) {
            if(fiveK_count + tenK_count < 100){
                writeRecord(SATURDAY_ROSTER, satEntry);
                if (satRace.equals("5k")) {
                    fiveK_count++;
                } else {
                    tenK_count++;
                }
                System.out.println("Adding entry to " + SATURDAY_ROSTER);
            } else {
                System.out.println("Saturday race is full. Cannot add entry.");
            }
        } else {
            System.out.println("Already registered for a saturday race.");
        }
        if (sunEntry != null && !isDuplicate(sunEntry, false)) {
            if(half_count + full_count < 100){
                writeRecord(SUNDAY_ROSTER, sunEntry);
                if (sunRace.equals("half")) {
                    half_count++;
                } else {
                    full_count++;
                }
                System.out.println("Adding entry to " + SUNDAY_ROSTER);
            } else {
                System.out.println("Sunday race is full. Cannot add entry.");
            }
        } else {
            System.out.println("Already registered for a sunday race.");
        }
    }
    public static String setRoster(String raceCat) {
        switch (raceCat) {
            case "5k":
                return _5K_ROSTER_FILE;
            case "10k":
                return _10K_ROSTER_FILE;
            case "half":
                return HALF_ROSTER_FILE;
            case "full":
                return FULL_ROSTER_FILE;
            default:
                return null;
        }
    }

    // Static initializer: initialize the four counters by reading each file's last
line.
    static {
        try {
            fiveK_count = readLastLineLeadingInt(_5K_ROSTER_FILE);
        } catch (Exception e) {
            fiveK_count = 0;
        }
        try {
            tenK_count = readLastLineLeadingInt(_10K_ROSTER_FILE);
        } catch (Exception e) {
            tenK_count = 0;
        }
        try {
            half_count = readLastLineLeadingInt(HALF_ROSTER_FILE);
        } catch (Exception e) {
            half_count = 0;
        }
        try {
            full_count = readLastLineLeadingInt(FULL_ROSTER_FILE);
        } catch (Exception e) {
            full_count = 0;
        }
    }

    /**
     * Read only the last logical line of the given file and return the leading integer
token.
```

```
     * The file is expected to have delimited records whose first token is an integer
sequence
     * or count. If the file does not exist or is empty, 0 is returned. If the last
line's
     * leading token is not a parsable integer, a NumberFormatException is thrown.
     */
    public static int readLastLineLeadingInt(String filename) throws IOException {
        File f = new File(filename);
        if (!f.exists()) return 0;

        try (java.io.RandomAccessFile raf = new java.io.RandomAccessFile(f, "r")) {
            long length = raf.length();
            if (length == 0) return 0;

            long pos = length - 1;
            // Skip trailing newlines/carriage returns
            while (pos >= 0) {
                raf.seek(pos);
                int b = raf.read();
                if (b == '\n' || b == '\r') { pos--; continue; }
                break;
            }

            if (pos < 0) return 0;

            // Read bytes backwards until previous newline or start of file
            java.io.ByteArrayOutputStream baos = new java.io.ByteArrayOutputStream();
            while (pos >= 0) {
                raf.seek(pos);
                int b = raf.read();
                if (b == '\n') break;
                baos.write(b);
                pos--;
            }

            byte[] rev = baos.toByteArray();
            // reverse to get correct order
            for (int i = 0, j = rev.length - 1; i < j; i++, j--) {
                byte t = rev[i]; rev[i] = rev[j]; rev[j] = t;
            }

            String lastLine = new String(rev,
java.nio.charset.StandardCharsets.UTF_8).trim();
            if (lastLine.endsWith("\r")) lastLine = lastLine.substring(0,
lastLine.length() - 1);
```

```java
        if (lastLine.isEmpty()) return 0;

        // first token before the file delimiter
        String firstToken = lastLine.split(DELIM_REGEX, 2)[0].trim();
        return Integer.parseInt(firstToken);
    }
}

/**
 * Parse an input string that may be either a date (yyyyMMdd) or a
 * timestamp (yyyyMMddHHmmss) and return the corresponding LocalDate.
 * If parsing both patterns fails but the input contains at least 8 digits,
 * the first 8 digits are used as the date portion.
 *
 * Throws DateTimeParseException if no valid date can be extracted.
 */
public static LocalDate parseToLocalDate(String input) {
    String s = input == null ? "" : input.trim();
    try {
        return LocalDate.parse(s, DOB_FMT);
    } catch (DateTimeParseException e) {
        // Fall through and try timestamp
    }

    // Try yyyyMMddHHmmss as LocalDateTime then convert
    try {
        LocalDateTime dt = LocalDateTime.parse(s, REG_TS_FMT);
        return dt.toLocalDate();
    } catch (DateTimeParseException e) {
        // Fall through to heuristic
    }

    String digits = s.replaceAll("\\D", "");
    if (digits.length() >= 8) {
        String datePart = digits.substring(0, 8);
        return LocalDate.parse(datePart, DOB_FMT);
    }

    throw new DateTimeParseException("Unparseable date: " + input, input, 0);
}

public static int writeRecord(String rosterFile, String entry) throws IOException {
    try (FileWriter fw = new FileWriter(rosterFile, true);
         PrintWriter pw = new PrintWriter(fw)) {
        pw.println(entry);
    }
    return 0;
}

/*
 * Newest version of isDuplicate
 */
private static boolean isDuplicate(String entry, boolean sat) throws IOException {
    String[] entryFields = entry.split(DELIM_REGEX, -1);

    // Build the target values using the configured indices.
    String[] target = new String[FIELDS_TO_COMPARE.length];
    for (int i = 0; i < FIELDS_TO_COMPARE.length; i++) {
        target[i] = entryFields[FIELDS_TO_COMPARE[i]].trim();
    }

    File[] files;
    if (sat) {
        files = new File[]{new File(_5K_ROSTER_FILE), new File(_10K_ROSTER_FILE)};
    } else {
        files = new File[]{new File(HALF_ROSTER_FILE), new File(FULL_ROSTER_FILE)};
    }

    // Process each roster file individually.
    for (File file : files) {
        if (!file.exists()) {   continue; }

        try (BufferedReader br = new BufferedReader(new FileReader(file))) {
            String line;

            while ((line = br.readLine()) != null) {
                String[] recordFields = line.split(DELIM_REGEX, -1);
                boolean matches = true;
                for (int i = 0; i < FIELDS_TO_COMPARE.length; i++) {
                    int fieldIndex = FIELDS_TO_COMPARE[i];
                    if (fieldIndex >= recordFields.length) {
                        matches = false;
                        break;
                    }
                    if (!recordFields[fieldIndex].trim().equals(target[i])) {
                        matches = false;
                        break;
                    }
                }
                if (matches) { return true; }
            }
        }
    }
    return false;
}
}
```