Franklin Marathon Test-Driven Development Plan

TDD Requirement 3.1:

Given a registration timestamp and a race distance, the system shall calculate the correct price for the race using the following table:

Period	5K	10K	Half	Full
Super Early	$30	$50	$65	$75
Early	$40	$55	$70	$80
Baseline	$50	$70	$85	$95
Late	$64	$89	$99	$109

Test Case 1

Input	20251030 5k
Expected Output	30
Actual Output	30
Pass/Fail	Pass

Test Case 2

Input	20251030 10k
Expected Output	50
Actual Output	50
Pass/Fail	Pass

Test Case 3

Input	20251030 Half
Expected Output	65
Actual Output	65
Pass/Fail	Pass

Test Case 4

Input	20251030 Full
Expected Output	75
Actual Output	75
Pass/Fail	Pass

Test Case 5

Input	20251108 5k
Expected Output	40
Actual Output	40
Pass/Fail	Pass

Test Case 6

Input	20251108 10K
Expected Output	55
Actual Output	55
Pass/Fail	Pass

Test Case 7

Input	20251108 Half
Expected Output	70
Actual Output	70
Pass/Fail	Pass

Test Case 8

Input	20251108 Full
Expected Output	80
Actual Output	80
Pass/Fail	Pass

Test Case 9

Input	20260312 5k
Expected Output	50
Actual Output	50
Pass/Fail	Pass

Test Case 10

Input	20260312 10K
Expected Output	70
Actual Output	70
Pass/Fail	Pass

Test Case 11

Input	20260312 Half
Expected Output	85
Actual Output	85
Pass/Fail	Pass

Test Case 12

Input	20260312 Full
Expected Output	95
Actual Output	95
Pass/Fail	Pass

Test Case 13

Input	20260415 5k
Expected Output	64
Actual Output	64
Pass/Fail	Pass

Test Case 14

Input	20260415 10K
Expected Output	89
Actual Output	89
Pass/Fail	Pass

Test Case 15

Input	20260415 Half
Expected Output	99
Actual Output	99
Pass/Fail	Pass

Test Case 16

Input	20260415 Full
Expected Output	109
Actual Output	109
Pass/Fail	Pass

```java
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
import java.util.Scanner;
import java.util.Map;
import java.util.HashMap;

public class req31 {
    private static final DateTimeFormatter INPUT_FMT =
DateTimeFormatter.ofPattern("yyyyMMdd");

    private static final Map<String, Map<String, Integer>> FEE_MAP = new HashMap<>();
    static {
        FEE_MAP.put("5k", Map.of(
                "super early", 30,
                "early", 40,
                "baseline", 50,
                "late", 64
        ));
        FEE_MAP.put("10k", Map.of(
                "super early", 50,
                "early", 55,
                "baseline", 70,
                "late", 89
        ));
        FEE_MAP.put("half", Map.of(
                "super early", 65,
                "early", 70,
                "baseline", 85,
                "late", 99
        ));
        FEE_MAP.put("full", Map.of(
                "super early", 75,
                "early", 80,
                "baseline", 95,
                "late", 109
        ));
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter date of registration (YYYYMMDD): ");
        LocalDate dor = LocalDate.parse(sc.next().trim(), INPUT_FMT);
        System.out.print("Enter race category: ");
        String raceCategory = sc.next().trim().toLowerCase();
```

```
        sc.close();

        String period = req11.determineRacePeriod(dor).trim().toLowerCase();
        int fee = determineFee(raceCategory, period);
        System.out.println("Registration period: " + period + ", Fee: $" + fee);
    }

    public static int determineFee(String cat, String per) {
        cat = cat.trim().toLowerCase();
        per = per.trim().toLowerCase();
        if (cat == null || per == null) return 0;

        Map<String, Integer> catMap = FEE_MAP.get(cat);
        if (catMap == null) return 0;
        return catMap.getOrDefault(per, 0);
    }
}
```

TDD Requirement 3.2:

The system shall allow a runner to sign up for any one of four races (5K, 10K, Half Marathon, and Full Marathon) and add the runner's information to the correct roster.

	Test Case 1
Input	DOB: 19601030 Registration: 20251029 5k
Expected Output	30
Actual Output	30
Pass/Fail	Pass

	Test Case 2
Input	DOB: 19601030 Registration: 20251031 5k
Expected Output	25 (30 -5)
Actual Output	25 (30 -5)
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.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class req32 {
    private static final DateTimeFormatter DOB_FMT =
DateTimeFormatter.ofPattern("yyyyMMdd");
    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};

    // How many lines/fields each record contains in the roster files.
    // Set this to the exact number of fields per record in the roster files
    // (for example 6 if records are: firstName, lastName, age, gender, email, regTs).
    private static final int RECORD_FIELDS_COUNT = 6;

    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 (YYYYMMDDHHMMSS) :");
        String regTs = sc.next().trim();
        System.out.print("Enter race category :");
```

```
        String raceCategory = sc.next().trim().toLowerCase();
        sc.close();

        LocalDate dob = LocalDate.parse(dobStr, DOB_FMT);
        LocalDate regDate = LocalDate.parse(regTs, DOB_FMT);

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

        //  TODO this will be adjusted later to determine age on the day of the
different races
        LocalDate raceDayShort = tday.plusDays(2);

        int ageOnRaceDay = req12.calculateAge(dob, raceDayShort);
        int ageOnRegister = req12.calculateAge(dob, regDate);
        int cost = req31.determineFee(raceCategory,
req11.determineRacePeriod(regDate));
        System.out.println("initial cost: $" + cost);
        cost = cost - (ageOnRegister > 64 ? 5 : 0);
        System.out.println("final cost: $" + cost);

        String entry = firstName + "\n" + lastName + "\n" + ageOnRaceDay + "\n" +
gender + "\n" + email + "\n" + regTs + "\n" + cost;

        String ROSTER_FILE;
        switch (raceCategory.toLowerCase()) {
            case "5k":
                ROSTER_FILE = _5K_ROSTER_FILE;
                break;
            case "10k":
                ROSTER_FILE = _10K_ROSTER_FILE;
                break;
            case "half":
                ROSTER_FILE = HALF_ROSTER_FILE;
                break;
            case "full":
                ROSTER_FILE = FULL_ROSTER_FILE;
                break;
            default:
                System.out.println("Invalid race category.");
                return;
        }

        if (!isDuplicate(entry)) {
            try (FileWriter fw = new FileWriter(ROSTER_FILE, true);
```

```
                 PrintWriter pw = new PrintWriter(fw)) {
                pw.println(firstName);
                pw.println(lastName);
                pw.println(ageOnRaceDay);
                pw.println(gender);
                pw.println(email);
                pw.println(regTs);
                pw.println(cost);
            }
            System.out.println(firstName);
            System.out.println(lastName);
            System.out.println(ageOnRaceDay);
            System.out.println(gender);
            System.out.println(email);
            System.out.println(regTs);
            System.out.println(raceCategory);
            System.out.println("$" + cost);
        } else {
            System.out.println("Duplicate entry. Not added to roster.");
        }
    }

    private static boolean isDuplicate(String entry) throws IOException {
        // Parse the incoming entry into fields (one field per line)
        String[] entryFields = entry.split("\n");

        // Verify the provided entry contains all indices we plan to compare.
        int maxNeededIndex = 0;
        for (int idx : FIELDS_TO_COMPARE) {
            if (idx > maxNeededIndex) {
                maxNeededIndex = idx;
            }
        }

        // 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]];
        }

        File[] files = {new File(_5K_ROSTER_FILE), new File(_10K_ROSTER_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))) {
                List<String> record = new ArrayList<>();
                String line;

                // Read file line-by-line and group records by fixed
RECORD_FIELDS_COUNT.
                while ((line = br.readLine()) != null) {
                    record.add(line);

                    // Once we've collected RECORD_FIELDS_COUNT lines, treat them as a
record.
                    if (record.size() == RECORD_FIELDS_COUNT) {
                        boolean matches = true;
                        for (int i = 0; i < FIELDS_TO_COMPARE.length; i++) {
                            int fieldIndex = FIELDS_TO_COMPARE[i];
                            // If the requested index is out of bounds for this record,
it's not a match.
                            if (fieldIndex >= record.size()) {
                                matches = false;
                                break;
                            }
                            if (!record.get(fieldIndex).equals(target[i])) {
                                matches = false;
                                break;
                            }
                        }
                        if (matches) {  return true; }
                        // Reset buffer to start collecting next record
                        record.clear();
                    }
                }
            }
        }
        return false;
    }
}
```