TDD Requirement 1.1:

Given a date stamp in the format of YYYYMMDD, the system shall be able to determine the location of that time within the race calendar year, according to the following table:
where TDay is the Thursday before the first Saturday of May.

Dates	Race Calendar Period
1 Jun – 30 Sep	Registration Not Open
Oct 1 – Oct 31	Super Early
Nov 1 – Feb 28/29	Early
Mar 1 – Apr 1	Baseline
Apr 2 – TDay	Late
TDay – 31 May	Registration Closed

Test Case 1

Input: 20250930
Expected Output: Registration Not Open
Actual Output: Registration Not Open
Pass/Fail: Pass

Test Case 2

Input: 20251001
Expected Output: Super Early
Actual Output: Super Early
Pass/Fail: Pass

Test Case 3

Input: 20251031
Expected Output: Super Early
Actual Output: Super Early
Pass/Fail: Pass

Test Case 4

Input: 20251101
Expected Output: Early
Actual Output: Early
Pass/Fail: Pass

Test Case 5

Input: 20260228
Expected Output: Early
Actual Output: Early
Pass/Fail: Pass

Test Case 6

Input: 20260301
Expected Output: Baseline
Actual Output: Baseline
Pass/Fail: Pass

Test Case 7

Input: 20260401
Expected Output: Baseline
Actual Output: Baseline
Pass/Fail: Pass

Test Case 8

Input: 20260402
Expected Output: Late
Actual Output: Late
Pass/Fail: Pass

Test Case 9

Input: 20260430
Expected Output: Late
Actual Output: Late
Pass/Fail: Pass

Test Case 10

Input: 20250501
Expected Output: Registration Not Open
Actual Output: Registration Not Open
Pass/Fail: Pass

TDD Requirement 1.2:

Given a date of birth, the system shall be able to calculate the age of a runner on race day. (Race day is TDay + 2 for the 5K and 10K, and TDay + 3 for the full and half marathons.)

Test Case 1

Input: 19980502
Expected Output: 27 sat, 27 sun
Actual Output: 27 sat, 27 sun
Pass/Fail: Pass

Test Case 2

Input: 19980503
Expected Output: 26 sat, 27 sun
Actual Output: 26 sat, 27 sun
Pass/Fail: Pass

TDD Requirement 2.1:

Given the information in the column on the left, the system shall add a runner to the race roster that includes the information in the column on the right

Info from Registration	Info Stored in Race Roster
a. First Name	a. First Name
b. Last Name	b. Last Name
c. Date of Birth	c. Date of Birth
d. Gender	d. Gender
e. Email Address	e. Email Address
f. Registration Timestamp	f. Registration Timestamp

Test Case 1				Pass/Fail	Pass
Input	First: John	Expected Output	First: John	Actual Output	First: John
	Last: Smith		Last: Smith		Last: Smith
	DOB: 2000-05-20		Age: 25		Age: 25
	Gender: M		Gender: M		Gender: M
	Email: john.smith@email.com		Email: john.smith@email.com		Email: john.smith@email.com
Test Case 2				Pass/Fail	Pass
Input	First: John	Expected Output	Duplicate entry. Not added to roster.	Actual Output	Duplicate entry. Not added to roster.
	Last: Smith				
	DOB: 2000-05-20				
	Gender: M				
	Email: john.smith@email.com				
Test Case 3				Pass/Fail	Pass
Input	First: Sarah	Expected Output	First: Sarah	Actual Output	First: Sarah
	Last: Jones		Last: Jones		Last: Jones
	DOB: 2001-05-20		Age: 24		Age: 24
	Gender: F		Gender: F		Gender: F
	Email: sarah.jones@email.com		Email: sarah.jones@email.com		Email: sarah.jones@email.com

TDD Requirement 2.2:

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				**Pass/Fail**	Pass
Input	First: John	**Expected Output**	First: John	**Actual Output**	First: John
	Last: Smith		Last: Smith		Last: Smith
	DOB: 2000-05-20		Age: 25		Age: 25
	Gender: M		Gender: M		Gender: M
	Email: john.smith@email.com		Email: john.smith@email.com		Email: john.smith@email.com
	Race: 5k		Race: 5k		Race: 5k
Test Case 2				**Pass/Fail**	Pass
Input	First: John	**Expected Output**	Duplicate entry. Not added to roster.	**Actual Output**	Duplicate entry. Not added to roster.
	Last: Smith				
	DOB: 2000-05-20				
	Gender: M				
	Email: john.smith@email.com				
	Race: 5k				
Test Case 3				**Pass/Fail**	Pass
Input	First: Sarah	**Expected Output**	First: Sarah	**Actual Output**	First: Sarah
	Last: Jones		Last: Jones		Last: Jones
	DOB: 2001-05-20		Age: 24		Age: 24
	Gender: F		Gender: F		Gender: F
	Email: sarah.jones@email.com		Email: sarah.jones@email.com		Email: sarah.jones@email.com
	Race: 10k		Race: 10k		Race: 10k
Test Case 4				**Pass/Fail**	Pass
Input	James	**Expected Output**	James	**Actual Output**	James
	O'Connor		O'Connor		O'Connor
	1985-06-01		Age: 40		Age: 40
	M		M		M
	james.oconnor@email.com		james.oconnor@email.com		james.oconnor@email.com
	Half Marathon		Half Marathon		Half Marathon

Test Case 5				Pass/Fail	Pass
Input	Olivia	Expected Output	Olivia	Actual Output	Olivia
	Davis		Davis		Davis
	1997-11-30		Age: 28		Age: 28
	F		F		F
	olivia.davis@email.com		olivia.davis@email.com		olivia.davis@email.com
	Full Marathon		Full Marathon		Full Marathon

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

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

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 Bundle with no Senior Discount	
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

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	
Precondition	100 Sunday runners
Input	Half
Expected Output	Sunday rosters full.
Actual Output	Sunday rosters full.
Pass/Fail	Pass
Verify denial when @100	
Precondition	100 Saturday runners
Input	5k
Expected Output	Saturday rosters full.
Actual Output	Saturday rosters full.
Pass/Fail	Pass

TDD Requirement 5.1:
The system shall be able to print off rosters for all races, showing these fields for each runner:

Info Stored in Race Roster
a. Last Name
b. First Name
c. Date of Birth
d. Age on Race Day
e. Division
f. Email Address
g. Registration Date & Amt Paid

At the bottom of each roster, the system will print the number of runners who have signed up for that event.

Verify ability to print rosters	
Precondition	At least ~5 names on roster
Input	5k Roster
Expected Output	Roster printout
Actual Output	Roster printed
Pass/Fail	Pass
Verify ability to print rosters	
Precondition	At least ~5 names on roster
Input	Half-Marathon Roster
Expected Output	Roster printout
Actual Output	Roster printed
Pass/Fail	Pass
Verify ability to print rosters	
Precondition	At least ~5 names on roster
Input	10k Roster
Expected Output	Roster printout
Actual Output	Roster printed
Pass/Fail	Pass
Verify ability to print rosters	
Precondition	At least ~5 names on roster
Input	Full-Marathon Roster
Expected Output	Roster printout
Actual Output	Roster printed
Pass/Fail	Pass

Franklin Marathon Acceptance Plan

20250601	Registration Not Open				
20250930	Registration Not Open				
20251001	Super Early		Legend for Color Code		
20251031	Super Early		Req 1.1	Tday/Reg Period	
20251101	Early		Req 1.2	Age on raceday	
20260228	Early		Req 2.1 & 2	4 Rosters and fields	
20260301	Baseline		Req 3.1, 2 & 4.1	Pricing & Discounts	
20260401	Baseline		Req 4.2	Capacity	
20260402	Late		Req 5.1	Print	
20260430	Late		Covered Tests		
20260501	Registration Closed				
20260531	Registration Closed				
	Without Senior Discount				
	5k	10k	Half	Full	
Super Early	30	50	65	75	
Early	40	55	70	80	
Base	50	70	85	95	
Late	64	89	99	109	
	With Senior Discount				
	5k	10k	Half	Full	
Super Early	25	45	60	70	
Early	35	50	65	75	
Base	45	65	80	90	
Late	59	84	94	104	
	Without Senior Discount				
	5k/half	5k/full	10k/half	10k/full	
Super Early	76	84	92	100	
Early	88	96	100	108	
Base	108	116	124	132	
Late	130.4	138.4	150.4	158.4	
	With Senior Discount				
	5k/half	5k/full	10k/half	10k/full	
Super Early	66	74	82	90	
Early	78	86	90	98	
Base	98	106	114	122	
Late	120.4	128.4	140.4	148.4	
Test Case 1				**Pass/Fail**	Pass
Input	20250930	**Expected Output**	Registration Not Open	**Actual Output**	Registration Not Open
Purpose	Test registration is not open.				
Test Case 2				**Pass/Fail**	Pass
Input	Scott	**Expected Output**	Scott	**Actual Output**	Scott
	Denlinger		Denlinger		Denlinger
	19590502		67/67		67/67
	M		M		M
	denlingers@hotmail.com		denlingers@hotmail.com		denlingers@hotmail.com
	5k and Half		5k and Half		5k and Half
	20251031		20251015		20251015

	66 [(30 + 65) * 0.8 – 10]		66 [(30 + 65) * 0.8 – 10]		66 [(30 + 65) * 0.8 – 10]
Purpose	Test birthday day of first race, 5k and half rosters and pricing, with Super Early registration, bundle and senior discount				
Test Case 3				**Pass/Fail**	Pass
Input	John	**Expected Output**	John	**Actual Output**	John
	Smith		Smith		Smith
	19990503		26/27		26/27
	M		M		M
	john.smith@email.com		john.smith@email.com		john.smith@email.com
	5k and Full		5k and Full		5k and Full
	20260228		20260228		20260228
	96 [(40 + 80) * 0.8]		96 [(40 + 80) * 0.8]		96 [(40 + 80) * 0.8]
Purpose	Test birthday day of second raceday, 5k and full rosters and pricing, with early registration and bundle				
Test Case 4				**Pass/Fail**	Pass
Input	Sydney	**Expected Output**	Sydney	**Actual Output**	Sydney
	Walker		Walker		Walker
	19990504		26/26		26/26
	F		F		F
	Sydneyw@yahoo.com		Sydneyw@yahoo.com		Sydneyw@yahoo.com
	10k and Half		10k and Half		10k and Half
	20260401		20260312		20260312
	124 [(70 + 85) * 0.8]		124 [(70 + 85) * 0.8]		124 [(70 + 85) * 0.8]
Purpose	Test birthday after races, 10k and half rosters and pricing, with base registration and bundle				
Test Case 5				**Pass/Fail**	Pass
Input	Olivia	**Expected Output**	Olivia	**Actual Output**	Olivia
	Davis		Davis		Davis
	19610502		65		65
	F		F		F
	olivia.davis@email.com		olivia.davis@email.com		olivia.davis@email.com
	10k and Full		10k and Full		10k and Full
	20260430		20260430		20260430
	158.4 [(89 + 109) * 0.8]		158.4 [(89 + 109) * 0.8]		158.4 [(89 + 109) * 0.8]
Purpose	Test 10k and full rosters and pricing with late registration and bundle, with 65th birthday between registration and raceday				
Test Case 6				**Pass/Fail**	Pass
Input	James	**Expected Output**	James	**Actual Output**	James
	Connor		Connor		Connor
	19610415		65		65
	M		M		M
	james.connor@email.com		james.connor@email.com		james.connor@email.com
	Half		Half		Half
	20251101		20251101		20251101
	65 (70 – 5)		65 (70 – 5)		65 (70 – 5)
Purpose	Test single race pricing with senior discount and early registration				
Test Case 7				**Pass/Fail**	Pass
Input	Sarah	**Expected Output**	Sarah	**Actual Output**	Sarah
	Jones		Jones		Jones
	20010520		24		24
	F		F		F
	sarah.jones@email.com		sarah.jones@email.com		sarah.jones@email.com
	10k		10k		10k
	20260301		20260301		20260301
	70		70		70
Purpose	Test single race pricing without senior discount and base registration				

Test Case 8				Pass/Fail	Pass
Input	20260501	**Expected Output**	Registration Closed	**Actual Output**	Registration Closed
Purpose	Test registration closed.				
Test Case 9				**Pass/Fail**	Pass
Preconditions	Small script used to fill rosters with User1, User2 etc				
Input	Kristi Herzog 20000501 F Hersogk@gmail.com 5k and Full 20260402 None	**Expected Output**	Saturday rosters full Sunday rosters full	**Actual Output**	Saturday rosters full Sunday rosters full
Purpose	Test capacity per day				
Test Case 10				**Pass/Fail**	Pass
Preconditions		All previous tests have been performed			
Input	Print rosters	**Expected Output**	Print rosters	**Actual Output**	Print rosters
Purpose	Test printing rosters				

```
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.DayOfWeek;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.Month;
import java.time.Period;
import java.time.format.DateTimeFormatter;
import java.time.format.DateTimeParseException;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;

public class project3 {
    private static final DateTimeFormatter DOB_FMT = DateTimeFormat-
ter.ofPattern("yyyyMMdd");
    private static final DateTimeFormatter REG_TS_FMT = DateTimeFormat-
ter.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.
```

```java
    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(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 = "\\|";

    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) throws IOException {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter command: ");
        String input = sc.next().trim();
        while (!input.equalsIgnoreCase("exit")) {
```

```
            if (input.equalsIgnoreCase("register")) {
                new project3().registerRunner(sc);
            } else if (input.equalsIgnoreCase("display")) {
                new project3().displayRoster(sc);
            } else if (!input.equalsIgnoreCase("exit")) {
                System.out.println("Unknown command. Valid commands are: register, dis-
play, exit.");
            }
            System.out.print("Enter command: ");
            input = sc.next().trim();
        }
        sc.close();
    }

    private void registerRunner(Scanner scan) throws IOException {
        System.out.print("Enter first name :");
        String firstName = scan.next().trim();
        System.out.print("Enter last name :");
        String lastName = scan.next().trim();
        System.out.print("Enter date of birth (YYYYMMDD) :");
        String dobStr = scan.next().trim();
        System.out.print("Enter gender :");
        String gender = scan.next().trim();
        System.out.print("Enter email :");
        String email = scan.next().trim();
        System.out.print("Enter registration timestamp (YYYYMMDD or YYYYMMDDHHMMSS)
:");
        String regTs = scan.next().trim();
        scan.nextLine();
        System.out.print("Enter Saturday race category (5k or 10k) :");
        String satRace = scan.nextLine().trim().toLowerCase();
        System.out.print("Enter Sunday race category (half or full) :");
        String sunRace = scan.nextLine().trim().toLowerCase();

        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 = computeTDay(raceYear);
        LocalDate satRaceDate = tday.plusDays(2);
        LocalDate sunRaceDate = tday.plusDays(3);

        int satRaceAge = calculateAge(dob, satRaceDate);
```

```
        int sunRaceAge = calculateAge(dob, sunRaceDate);
        int ageOnRegister = calculateAge(dob, regDate);
        int satCost = !satRace.isEmpty() ? determineFee(satRace, determineRacePeri-
od(regDate)) : 0;
        int sunCost = !sunRace.isEmpty() ? determineFee(sunRace, determineRacePeri-
od(regDate)) : 0;
        double cost = (satCost != 0 && sunCost != 0) ? (satCost + sunCost) * 0.8 -
(ageOnRegister > 64 ? 10 : 0)
                                                     : satCost + sunCost - (ageOnRegis-
ter > 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 cor-
ruption
        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 (!determineRacePeriod(regDate).equals("Registration Closed") && !determin-
eRacePeriod(regDate).equals("Registration Not Open")) {
            if (satEntry != null) {
                if(isDuplicate(satEntry, true)){
                    System.out.println("Already registered for a saturday race.");
                } else 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.");
            }
        }
        if (sunEntry != null) {
            if(isDuplicate(sunEntry, false)){
                System.out.println("Already registered for a sunday race.");
            } else 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(determineRacePeriod(regDate));
    }
}

private void displayRoster(Scanner sc) {
    System.out.print("Enter roster to print :");
    String roster = sc.next().trim();

    String ROSTER_FILE = setRoster(roster);

    if (ROSTER_FILE != null) {
        RosterPrinter.printRoster(ROSTER_FILE);
    } else {
        System.out.println("Invalid roster type entered.");
    }
}

public static String determineRacePeriod(LocalDate d) {
    int raceYear = d.getYear() + (d.getMonthValue() >= 6 ? 1 : 0);

    LocalDate superEarly = LocalDate.of(raceYear - 1, Month.OCTOBER, 1);
    LocalDate early = LocalDate.of(raceYear - 1, Month.NOVEMBER, 1);
```

```
        LocalDate baseline = LocalDate.of(raceYear, Month.MARCH, 1);
        LocalDate late = LocalDate.of(raceYear, Month.APRIL, 2);
        LocalDate registrationEnd = LocalDate.of(raceYear, Month.MAY, 31);

        if(d.isBefore(superEarly)){
            return "Registration Not Open";
        }else if(d.isBefore(early)){
            return "Super Early";
        }else if(d.isBefore(baseline)){
            return "Early";
        }else if(d.isBefore(late)){
            return "Baseline";
        }else if(d.isBefore(registrationEnd) || d.isEqual(registrationEnd)){
            return "Late";
        } else {
            return "Registration Closed";
        }
    }

    public static LocalDate computeTDay(int year) {
        LocalDate firstOfMay = LocalDate.of(year, Month.MAY, 1);
        DayOfWeek dow = firstOfMay.getDayOfWeek();
        int daysToAdd = (DayOfWeek.SATURDAY.getValue() - dow.getValue() + 7) % 7;
        LocalDate firstSaturday = firstOfMay.plusDays(daysToAdd);
        return firstSaturday.minusDays(2);
    }

    public static int calculateAge(LocalDate dob, LocalDate onDate) {
        return Period.between(dob, onDate).getYears();
    }

    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);
    }

    public static String setRoster(String raceCat) {
        switch (raceCat.toLowerCase()) {
            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);
    }

    /*
```

```java
     * 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;
    }
```

```
    // 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, ja-
va.nio.charset.StandardCharsets.UTF_8).trim();
            if (lastLine.endsWith("\r")) lastLine = lastLine.substring(0, last-
Line.length() - 1);
            if (lastLine.isEmpty()) return 0;

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

    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;
    }
```

```
    public static class RosterPrinter {
        private static final String DELIM_REGEX = "\\|";

        public static void printRoster(String rosterFile) {
            if (rosterFile == null) {
                System.out.println("No roster file specified.");
                return;
            }

            File f = new File(rosterFile);
            if (!f.exists()) {
                System.out.println("Roster file not found: " + rosterFile);
                return;
            }

            int count = 0;
            try (BufferedReader br = new BufferedReader(new FileReader(f))) {
                // Print header row for table: Last, First, Age & Sex, Email, Reg Date,
Cost
                System.out.printf("%-15s %-12s %-9s %-32s %-12s %8s%n", "Last",
"First", "Age/Sex", "Email", "Reg Date", "Cost");
                System.out.println(new String(new char[90]).replace('\0', '-'));

                String line;
                while ((line = br.readLine()) != null) {
                    if (line.trim().isEmpty()) continue;

                    String[] fields = line.split(DELIM_REGEX, -1);

                    String first = fields.length > 1 ? fields[1].trim() : "";
                    String last = fields.length > 2 ? fields[2].trim() : "";
                    String age = fields.length > 3 ? fields[3].trim() : "";
                    String email = fields.length > 5 ? fields[5].trim() : "";
                    String regTs = fields.length > 6 ? fields[6].trim() : "";
                    String amt = fields.length > 7 ? fields[7].trim() : "";

                    // Try to find a DOB-like field (yyyyMMdd). Prefer any 8-digit
field that is not the regTs field.
                    String dobField = "";
                    for (int i = 0; i < fields.length; i++) {
                        String fld = fields[i].trim();
                        if (fld.matches("\\d{8}") && (regTs.isEmpty() ||
!fld.equals(regTs.substring(0, Math.min(8, regTs.length()))))) {
                            dobField = fld;
                            break;
```

```
                    }
                }

                // If we didn't find an explicit DOB, try to approximate using
regTs and age (regDate - age years)
                if (dobField.isEmpty() && !regTs.isEmpty() && !age.isEmpty()) {
                    try {
                        LocalDate regDate = parseToLocalDate(regTs);
                        int a = Integer.parseInt(age);
                        LocalDate approxDob = regDate.minusYears(a);
                        dobField = String.format("%04d%02d%02d", approx-
Dob.getYear(), approxDob.getMonthValue(), approxDob.getDayOfMonth());
                    } catch (Exception e) {
                        // leave dobField empty if parse fails
                    }
                }

                // We will show Age & Sex together
                String sex = fields.length > 4 ? fields[4].trim() : "";
                String ageSex = (age.isEmpty() ? "" : age) + (sex.isEmpty() ? "" :
" " + sex);

                String regDateDisplay = regTs;
                if (!regTs.isEmpty()) {
                    try {
                        LocalDate rd = parseToLocalDate(regTs);
                        regDateDisplay = rd.toString();
                    } catch (Exception e) {
                        // leave regTs as-is if parsing fails
                    }
                }

                // Format amount as currency with two decimals
                String costDisplay = amt == null || amt.isEmpty() ? "0.00" : amt;
                try {
                    double v = Double.parseDouble(costDisplay);
                    costDisplay = String.format("$%.2f", v);
                } catch (NumberFormatException e) {
                    // keep as-is
                }

                // Print table row
                System.out.printf("%-15s %-12s %-9s %-32s %-12s %8s%n",
                    last, first, ageSex, email, regDateDisplay, costDisplay);
                count++;
```

```
                }

                // Footer line like sample
                System.out.println();
                System.out.println("There are " + count + " runners registered for this
race.");

            } catch (IOException e) {
                System.out.println("Error reading roster file: " + e.getMessage());
            }
        }
    }
}
```