Module 7 of 7

Expense sharing

Splitwise in Java: money in paise, equal, exact and percent splits as strategies, net balances that add up to zero, and the greedy settle-up.

An expense sharing app (Splitwise, in most rounds) is a ledger with one invariant: money is moved, never created, so a group's balances always add up to zero. The machine-coding problem tests whether the code keeps that invariant through every split rule, every rounding and every settle-up.

You will write it in three steps. First the split rules, equal, exact and percent, each a strategy that validates its own input and turns an amount into shares, over money kept in integer paise so nothing drifts. Then the settle-up: the greedy rule that has the largest debtor pay the largest creditor until everyone is square, in at most n - 1 payments. The checkpoint adds a fourth rule, shares by weight, to show what the Strategy buys you.

Balances are kept in name order, so every listing and every tie is deterministic. The full design is in the expense sharing note.

CONTENTS

1	Expense sharing: splits, balances and settling up	14 min
✓	Checkpoint: Expense sharing	25 min

Exercises in this module compile on OpenJDK 18. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Expense sharing: splits, balances and settling up

An expense app is a ledger with one invariant: **money is moved, never created**, so a group's balances always add up to zero. The round tests the split rules, the money arithmetic and the settle-up. The expense sharing note has the full design; this lesson is the code.

The classes

- **Member**: a name. **BalanceSheet** holds one net balance per member: positive means the group owes them.
- **Split** (interface): `int[] shares(long amount, …)`, one class per rule: `EqualSplit`, `ExactSplit`, `PercentSplit`. Each **validates its own input** before any balance moves. This is the Strategy pattern.
- **Expense**: payer, amount, split. Applying it: payer `+amount`, every share `-share`, the payer's own share included.

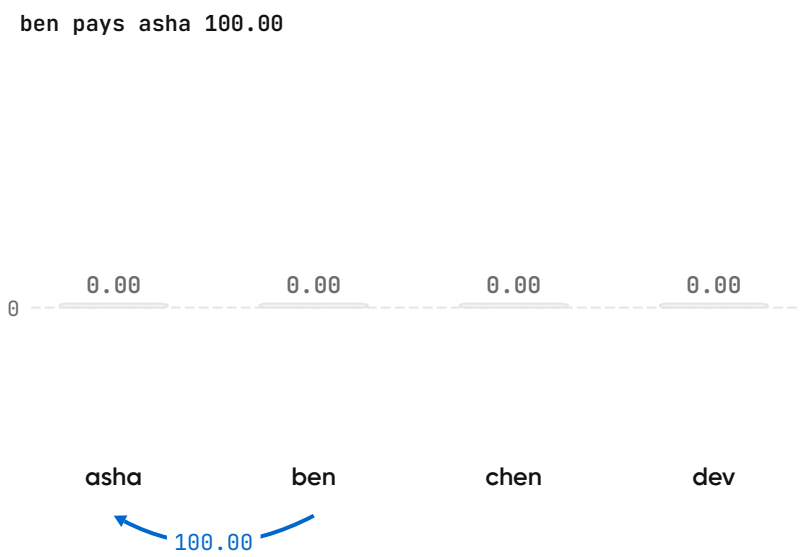
Money in paise

- `12.50` is read as `1250`, a `long`. Never a `double`: 0.1 has no exact binary form and the sums drift.
- An equal split of 100.00 among three is 33.34, 33.33, 33.33: the leftover paisa goes to the first member.
- A percent split floors each share; the leftover goes to the first member.
- Print with two decimals at the very end, and only there.

Settling up

1. Find the member owed the most and the member who owes the most (ties by name, so the output is fixed).
2. The debtor pays the smaller of the two amounts; at least one reaches zero.
3. Repeat until every balance is zero: at most n - 1 payments.

Settling up greedily: the largest debtor pays the largest creditor



1. Four friends, every balance at zero. A positive balance will mean the group owes that member; a negative one, that the member owes.
2. asha pays 1200.00 for all four. She gets +1200.00 and everyone, asha included, gets -300.00: asha ends at +900.00, the others at -300.00.
3. ben pays 200.00 for chen and dev. He moves up by 200.00 and they move down by 100.00 each. The bars still add up to zero: money moved, none was made.
4. Settle: the largest creditor is asha (+900.00), the largest debtors chen and dev tie at -400.00, and the earlier name goes first. chen pays the smaller amount, 400.00.
5. asha is still owed 500.00; dev owes the most, 400.00, and pays it all. dev is now square.
6. Only ben owes now, 100.00, exactly what asha is still owed. One payment settles both: three transfers for four people, within the n - 1 bound.

Greedy is not always the true minimum (that problem is NP-hard), but it is short, fast and never needs more than n - 1 transfers.

The traps

- **Pairwise debts.** Tracking "A owes B" for every pair is fine for display, but settling needs only the net balances.
- **Validation after the update.** An exact split that does not add up must be refused before any balance changes, or the ledger is left half-applied.
- **Rounding by accident.** `amount / n` drops paise; hand them out on purpose.

GAME · QUICK FIRE

Settle the group

Each round gives net balances in rupees. Pick the first payment the greedy rule makes.

1/4

asha +400, ben -100, chen -300

A chen pays asha 300

B ben pays asha 100

C chen pays asha 400

EXERCISE 1

Split strategies over paise

Record a group's expenses with three split rules and report the net balances.

Input. The first line lists the members' names. Amounts are rupees with at most two decimals (`900` , `10.5` , `40.25`); keep them as paise in a `long` . Then one command per line:

- `EXPENSE payer amount EQUAL u1 u2 …` : equal shares; leftover paise go one each to the first members listed.
- `EXPENSE payer amount EXACT u1 a1 u2 a2 …` : the given amounts, which must add up to the expense, else `exact shares add up to X, not Y` .
- `EXPENSE payer amount PERCENT u1 p1 u2 p2 …` : whole percentages that must total 100, else `percents add up to N, not 100` . Each share is floored to the paisa; the leftover goes to the first member listed.

  Check in this order: the payer is a member, else `unknown user NAME` ; the split type is known, else `unknown split TYPE` ; each participant is a member, else `unknown user NAME` ; then the split's own rule. A refused expense changes nothing. Otherwise the payer gets `+amount` and each participant `-share` (the payer may or may not be one of them), and the program prints `payer paid 900.00: asha 300.00, ben 300.00, chen 300.00` with the participants in the order given.
- `BALANCES` : every member in name order with their net balance, such as `asha -150.00` , `ben +250.00` , `chen 0.00` .

Example

```
kim lee max
EXPENSE kim 100 EQUAL kim lee max
BALANCES
```

prints

```
kim paid 100.00: kim 33.34, lee 33.33, max 33.33
kim +66.66
lee -33.33
max -33.33
```

INPUT 1

```
asha ben chen
EXPENSE asha 900 EQUAL asha ben chen
EXPENSE ben 1000 PERCENT asha 50 ben 20 chen 30
EXPENSE chen 600 EXACT asha 250 ben 250 chen 100
BALANCES
```

OUTPUT 1

```
asha paid 900.00: asha 300.00, ben 300.00, chen 300.00
ben paid 1000.00: asha 500.00, ben 200.00, chen 300.00
chen paid 600.00: asha 250.00, ben 250.00, chen 100.00
asha -150.00
ben +250.00
chen -100.00
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    /** "12", "12.5" or "12.50" rupees as paise; never through a double. */
    static long toPaise(String s) {
        int dot = s.indexOf('.');
        if (dot < 0) return Long.parseLong(s) * 100;
        String frac = (s.substring(dot + 1) + "00").substring(0, 2);
        return Long.parseLong(s.substring(0, dot)) * 100 + Long.parseLong(frac);
    }

    /** Paise as rupees with two decimals: 1250 → "12.50", -5 → "-0.05". */
    static String money(long p) {
        String sign = p < 0 ? "-" : "";
        p = Math.abs(p);
        return sign + (p / 100) + "." + String.format("%02d", p % 100);
    }

    static String signed(long p) { return p > 0 ? "+" + money(p) : money(p); }

    /** A split rule: the shares in paise, in the participants' order. Throws Illega
    interface Split {
        /** True when each participant is followed by a number (EXACT, PERCENT). */
        boolean takesValues();
        long[] shares(long amount, List<String> values);
    }

    static class EqualSplit implements Split {
        public boolean takesValues() { return false; }
        public long[] shares(long amount, List<String> values) {
            int n = values.size();
            long[] s = new long[n];
            for (int i = 0; i < n; i++) s[i] = amount / n + (i < amount % n ? 1 : 0)
            return s;
        }
    }

    static class ExactSplit implements Split {
        public boolean takesValues() { return true; }
        public long[] shares(long amount, List<String> values) {
            long[] s = new long[values.size()];
            long sum = 0;
            for (int i = 0; i < s.length; i++) { s[i] = toPaise(values.get(i)); sum
            if (sum ≠ amount) throw new IllegalArgumentException("exact shares add
            return s;
        }
    }
```

```
    static class PercentSplit implements Split {
        public boolean takesValues() { return true; }
        public long[] shares(long amount, List<String> values) {
            long[] s = new long[values.size()];
            int total = 0;
            for (String v : values) total += Integer.parseInt(v);
            if (total != 100) throw new IllegalArgumentException("percents add up to
            long sum = 0;
            for (int i = 0; i < s.length; i++) { s[i] = amount * Integer.parseInt(va
            s[0] += amount - sum; // rounding leftovers
            return s;
        }
    }

    static final Map<String, Split> SPLITS = Map.of("EQUAL", new EqualSplit(), "EXAC

    static class BalanceSheet {
        final TreeMap<String, Long> balance = new TreeMap<>(); // name order

        String expense(String[] p) {
            String payer = p[1];
            long amount = toPaise(p[2]);
            if (!balance.containsKey(payer)) return "unknown user " + payer;
            Split split = SPLITS.get(p[3]);
            if (split == null) return "unknown split " + p[3];
            List<String> names = new ArrayList<>(), values = new ArrayList<>();
            int step = split.takesValues() ? 2 : 1;
            for (int i = 4; i < p.length; i += step) {
                names.add(p[i]);
                values.add(step == 2 ? p[i + 1] : null);
            }
            for (String n : names) if (!balance.containsKey(n)) return "unknown user
            long[] s;
            try {
                s = split.shares(amount, values); // validated before any balance mo
            } catch (IllegalArgumentException e) {
                return e.getMessage();
            }
            balance.merge(payer, amount, Long::sum);
            StringJoiner line = new StringJoiner(", ", payer + " paid " + money(amou
            for (int i = 0; i < s.length; i++) {
                balance.merge(names.get(i), -s[i], Long::sum);
                line.add(names.get(i) + " " + money(s[i]));
            }
            return line.toString();
        }

        String balances() {
            StringJoiner j = new StringJoiner("\n");
```

```
            balance.forEach((name, b) -> j.add(name + " " + signed(b)));
            return j.toString();
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BalanceSheet sheet = new BalanceSheet();
        for (String name : br.readLine().trim().split("\\s+")) sheet.balance.put(nam
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            if (p[0].equals("EXPENSE")) out.append(sheet.expense(p)).append('\n');
            else if (p[0].equals("BALANCES")) out.append(sheet.balances()).append('\
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Settle up

Settle a group's balances with the greedy rule.

Input. The first line lists the members. Money is paise, as in the first program. Then one command per line:

- `EXPENSE payer amount EQUAL u1 u2 …` : an equal split, exactly as before (only `EQUAL` appears here), printing `payer paid 1200.00: asha 300.00, …` . An unknown payer or member prints `unknown user NAME` and changes nothing.
- `PAY from to amount` : `from` hands `to` money directly: `from` 's balance goes up by the amount and `to` 's goes down. Prints `from paid to 10.00` , or `unknown user NAME` (checking `from` first).
- `SETTLE` : while anyone is owed money, the member who owes the most pays the member owed the most the smaller of the two amounts; ties go to the earlier name. Print `debtor pays creditor 400.00` for each payment, then `transfers: N` . The payments are applied, so every balance is zero afterwards.
- `BALANCES` : every member in name order: `asha +150.00` , `ben -250.00` , `chen 0.00` .

Example

```
asha ben chen dev
EXPENSE asha 1200 EQUAL asha ben chen dev
EXPENSE ben 200 EQUAL chen dev
SETTLE
```

prints

```
asha paid 1200.00: asha 300.00, ben 300.00, chen 300.00, dev 300.00
ben paid 200.00: chen 100.00, dev 100.00
chen pays asha 400.00
dev pays asha 400.00
ben pays asha 100.00
transfers: 3
```

INPUT 1

```
asha ben chen dev
EXPENSE asha 1200 EQUAL asha ben chen
dev
EXPENSE ben 200 EQUAL chen dev
SETTLE
BALANCES
```

OUTPUT 1

```
asha paid 1200.00: asha 300.00, ben
300.00, chen 300.00, dev 300.00
ben paid 200.00: chen 100.00, dev 100.00
chen pays asha 400.00
dev pays asha 400.00
ben pays asha 100.00
transfers: 3
asha 0.00
ben 0.00
chen 0.00
dev 0.00
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    static long toPaise(String s) {
        int dot = s.indexOf('.');
        if (dot < 0) return Long.parseLong(s) * 100;
        String frac = (s.substring(dot + 1) + "00").substring(0, 2);
        return Long.parseLong(s.substring(0, dot)) * 100 + Long.parseLong(frac);
    }

    static String money(long p) {
        String sign = p < 0 ? "-" : "";
        p = Math.abs(p);
        return sign + (p / 100) + "." + String.format("%02d", p % 100);
    }

    static String signed(long p) { return p > 0 ? "+" + money(p) : money(p); }

    static class BalanceSheet {
        final TreeMap<String, Long> balance = new TreeMap<>(); // name order

        String expense(String payer, long amount, List<String> members) {
            if (!balance.containsKey(payer)) return "unknown user " + payer;
            for (String m : members) if (!balance.containsKey(m)) return "unknown us
            int n = members.size();
            balance.merge(payer, amount, Long::sum);
            StringJoiner line = new StringJoiner(", ", payer + " paid " + money(amou
            for (int i = 0; i < n; i++) {
                long share = amount / n + (i < amount % n ? 1 : 0);
                balance.merge(members.get(i), -share, Long::sum);
                line.add(members.get(i) + " " + money(share));
            }
            return line.toString();
        }

        String pay(String from, String to, long amount) {
            if (!balance.containsKey(from)) return "unknown user " + from;
            if (!balance.containsKey(to)) return "unknown user " + to;
            balance.merge(from, amount, Long::sum); // from has paid, so is owed mor
            balance.merge(to, -amount, Long::sum);
            return from + " paid " + to + " " + money(amount);
        }

        // Greedy: the largest debtor pays the largest creditor the smaller of the t
        String settle() {
            StringJoiner j = new StringJoiner("\n");
            int transfers = 0;
            while (true) {
```

```
                String cred = null, debt = null;
                for (Map.Entry<String, Long> e : balance.entrySet()) { // name order
                    long b = e.getValue();
                    if (b > 0 && (cred == null || b > balance.get(cred))) cred = e.g
                    if (b < 0 && (debt == null || b < balance.get(debt))) debt = e.g
                }
                if (cred == null) break; // balances sum to zero, so no creditor mec
                long pay = Math.min(balance.get(cred), -balance.get(debt));
                balance.merge(cred, -pay, Long::sum);
                balance.merge(debt, pay, Long::sum);
                j.add(debt + " pays " + cred + " " + money(pay));
                transfers++;
            }
            j.add("transfers: " + transfers);
            return j.toString();
        }

        String balances() {
            StringJoiner j = new StringJoiner("\n");
            balance.forEach((name, b) -> j.add(name + " " + signed(b)));
            return j.toString();
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BalanceSheet sheet = new BalanceSheet();
        for (String name : br.readLine().trim().split("\\s+")) sheet.balance.put(nam
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            String said = switch (p[0]) {
                case "EXPENSE" -> sheet.expense(p[1], toPaise(p[2]), Arrays.asList(p
                case "PAY" -> sheet.pay(p[1], p[2], toPaise(p[3]));
                case "SETTLE" -> sheet.settle();
                case "BALANCES" -> sheet.balances();
                default -> "unknown command " + p[0];
            };
            out.append(said).append('\n');
        }
        System.out.print(out);
    }
}
```

QUIZ

1. Why should an expense app store money as a `long` of paise rather than a `double` of rupees?

A. A long uses less memory

B. A double cannot represent amounts like 0.10 exactly, so sums drift and balances stop netting to zero

C. Java has no double arithmetic for money

D. Paise are what banks print on statements

2. An EQUAL split of 100.00 among three members gives which shares?

A. 33.33, 33.33, 33.33

B. 33.34, 33.33, 33.33

C. 33.33, 33.33, 33.34

D. 33.00, 33.00, 34.00

3. Where should "an exact split must add up to the amount" be checked?

A. In the ExactSplit strategy, before any balance changes

B. In BalanceSheet, after the balances are updated

C. In the main loop, by string matching

D. Nowhere; the user is trusted

4. Net balances are asha +500, ben -200, chen -300. What does the greedy settle-up print?

A. ben pays asha 200, then chen pays asha 300

B. chen pays asha 300, then ben pays asha 200

C. chen pays asha 500

D. asha pays chen 300, then asha pays ben 200

ANSWER KEY · EXPENSE SHARING: SPLITS, BALANCES AND SETTLING UP

1. B Binary floating point has no exact form for most decimal fractions, so 0.1 + 0.2 is not 0.3 and long ledgers accumulate error. Whole paise are exact; you format to rupees only when printing. (`BigDecimal` is the other correct answer in Java, at more cost.)

2. B 10000 paise / 3 is 3333 with 1 left over, and the rule hands leftover paise out one each from the first member. 33.33 three times loses a paisa, so the balances would not add up to zero.

3. A Each rule knows its own input, so ExactSplit validates its amounts, PercentSplit its percentages. Checking after the update leaves the ledger half-applied when the check fails; checking in the input loop scatters split rules outside the classes that own them.

4. B Each step takes the largest creditor (asha, 500) and the largest debtor (chen, -300); the smaller amount moves, so chen pays 300. Then ben, -200, pays the remaining 200. Two transfers for three people, within the n - 1 bound.

CHECKPOINT · 25 MIN

Checkpoint: Expense sharing

This checkpoint covers the module: split rules as strategies, money in paise, balances that always add up to zero, and the greedy settle-up.

How it works. Ten questions and one program. You need 70% on the questions and the program accepted to clear the module. Retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why is a `double` the wrong type for money, and what replaces it?
- Where do leftover paise go in an equal split, and in a percent split?
- When does a split refuse its input, and what must not happen before it does?
- Which two members does each greedy settle step pick, and how much moves?
- What changes in the code when a new split rule is added?

The program adds a fourth rule, shares by weight, to the three you have written. If the Split interface is right, it is one more class; if it is not, you will feel it now.

EXERCISE

A fourth split, and settle up

Extend the ledger with a fourth split rule and the settle-up.

Input. As in the lesson's first program: the members on the first line, then commands, money as paise.

- `EXPENSE payer amount TYPE …` with `TYPE` one of `EQUAL`, `EXACT`, `PERCENT` (exactly as before) or the new `SHARES u1 w1 u2 w2 …`: whole-number weights, each at least 1, else `shares must be positive`. Each share is `amount × w / total weight`, floored to the paisa, with the leftover to the first member listed. The checks and the printed line are as before.
- `BALANCES`: as before.
- `SETTLE`: while anyone is owed money, the member who owes the most pays the member owed the most the smaller of the two amounts. Ties go to the earlier name. Print `debtor pays creditor 400.00` for each payment and then `transfers: N`. The payments are applied, so every balance is zero afterwards.

Example

```
ira jo ken
EXPENSE ira 1200 SHARES ira 2 jo 1 ken 1
SETTLE
```

prints

```
ira paid 1200.00: ira 600.00, jo 300.00, ken 300.00
jo pays ira 300.00
ken pays ira 300.00
transfers: 2
```

INPUT 1

```
ira jo ken
EXPENSE ira 1200 SHARES ira 2 jo 1 ken 1
EXPENSE jo 300 EXACT ira 100 jo 50 ken
150
EXPENSE ken 100 PERCENT ira 25 jo 25 ken
50
BALANCES
SETTLE
```

OUTPUT 1

```
ira paid 1200.00: ira 600.00, jo 300.00,
ken 300.00
jo paid 300.00: ira 100.00, jo 50.00,
ken 150.00
ken paid 100.00: ira 25.00, jo 25.00,
ken 50.00
ira +475.00
jo -75.00
ken -400.00
ken pays ira 400.00
jo pays ira 75.00
transfers: 2
```

REFERENCE SOLUTION

```java
import java.io.*;
import java.util.*;

public class Main {
    /** "12", "12.5" or "12.50" rupees as paise; never through a double. */
    static long toPaise(String s) {
        int dot = s.indexOf('.');
        if (dot < 0) return Long.parseLong(s) * 100;
        String frac = (s.substring(dot + 1) + "00").substring(0, 2);
        return Long.parseLong(s.substring(0, dot)) * 100 + Long.parseLong(frac);
    }

    /** Paise as rupees with two decimals: 1250 → "12.50", -5 → "-0.05". */
    static String money(long p) {
        String sign = p < 0 ? "-" : "";
        p = Math.abs(p);
        return sign + (p / 100) + "." + String.format("%02d", p % 100);
    }

    static String signed(long p) { return p > 0 ? "+" + money(p) : money(p); }

    interface Split {
        boolean takesValues();
        long[] shares(long amount, List<String> values);
    }

    static class EqualSplit implements Split {
        public boolean takesValues() { return false; }
        public long[] shares(long amount, List<String> values) {
            int n = values.size();
            long[] s = new long[n];
            for (int i = 0; i < n; i++) s[i] = amount / n + (i < amount % n ? 1 : 0)
            return s;
        }
    }

    static class ExactSplit implements Split {
        public boolean takesValues() { return true; }
        public long[] shares(long amount, List<String> values) {
            long[] s = new long[values.size()];
            long sum = 0;
            for (int i = 0; i < s.length; i++) { s[i] = toPaise(values.get(i)); sum
            if (sum ≠ amount) throw new IllegalArgumentException("exact shares add
            return s;
        }
    }

    /** Shares by integer weights. PercentSplit is the special case where the weight
    static class WeightedSplit implements Split {
```

```
        public boolean takesValues() { return true; }
        void check(int[] w) {
            for (int x : w) if (x < 1) throw new IllegalArgumentException("shares mu
        }
        public long[] shares(long amount, List<String> values) {
            int[] w = new int[values.size()];
            int total = 0;
            for (int i = 0; i < w.length; i++) { w[i] = Integer.parseInt(values.get(
            check(w);
            long[] s = new long[w.length];
            long sum = 0;
            for (int i = 0; i < w.length; i++) { s[i] = amount * w[i] / total; sum +
            s[0] += amount - sum; // rounding leftovers
            return s;
        }
    }

    static class SharesSplit extends WeightedSplit { }

    static class PercentSplit extends WeightedSplit {
        @Override void check(int[] w) {
            int total = 0;
            for (int x : w) total += x;
            if (total != 100) throw new IllegalArgumentException("percents add up to
        }
    }

    static final Map<String, Split> SPLITS = Map.of(
        "EQUAL", new EqualSplit(), "EXACT", new ExactSplit(), "PERCENT", new Percent

    static class BalanceSheet {
        final TreeMap<String, Long> balance = new TreeMap<>(); // name order

        String expense(String[] p) {
            String payer = p[1];
            long amount = toPaise(p[2]);
            if (!balance.containsKey(payer)) return "unknown user " + payer;
            Split split = SPLITS.get(p[3]);
            if (split == null) return "unknown split " + p[3];
            List<String> names = new ArrayList<>(), values = new ArrayList<>();
            int step = split.takesValues() ? 2 : 1;
            for (int i = 4; i < p.length; i += step) {
                names.add(p[i]);
                values.add(step == 2 ? p[i + 1] : null);
            }
            for (String n : names) if (!balance.containsKey(n)) return "unknown user
            long[] s;
            try {
                s = split.shares(amount, values);
```

```
            } catch (IllegalArgumentException e) {
                return e.getMessage();
            }
            balance.merge(payer, amount, Long::sum);
            StringJoiner line = new StringJoiner(", ", payer + " paid " + money(amou
            for (int i = 0; i < s.length; i++) {
                balance.merge(names.get(i), -s[i], Long::sum);
                line.add(names.get(i) + " " + money(s[i]));
            }
            return line.toString();
        }

        String balances() {
            StringJoiner j = new StringJoiner("\n");
            balance.forEach((name, b) -> j.add(name + " " + signed(b)));
            return j.toString();
        }

        // Greedy: the largest debtor pays the largest creditor; strict comparisons
        String settle() {
            StringJoiner j = new StringJoiner("\n");
            int transfers = 0;
            while (true) {
                String cred = null, debt = null;
                for (Map.Entry<String, Long> e : balance.entrySet()) {
                    long b = e.getValue();
                    if (b > 0 && (cred == null || b > balance.get(cred))) cred = e.g
                    if (b < 0 && (debt == null || b < balance.get(debt))) debt = e.g
                }
                if (cred == null) break;
                long pay = Math.min(balance.get(cred), -balance.get(debt));
                balance.merge(cred, -pay, Long::sum);
                balance.merge(debt, pay, Long::sum);
                j.add(debt + " pays " + cred + " " + money(pay));
                transfers++;
            }
            j.add("transfers: " + transfers);
            return j.toString();
        }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BalanceSheet sheet = new BalanceSheet();
        for (String name : br.readLine().trim().split("\\s+")) sheet.balance.put(nam
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
```

```
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            String said = switch (p[0]) {
                case "EXPENSE" -> sheet.expense(p);
                case "BALANCES" -> sheet.balances();
                case "SETTLE" -> sheet.settle();
                default -> "unknown command " + p[0];
            };
            out.append(said).append('\n');
        }
        System.out.print(out);
    }
}
```

CHECKPOINT QUESTIONS

1. The product wants a split "by shares", such as 2 : 1 : 1. With a Split interface, what changes?

A. One new class implementing Split; Expense and BalanceSheet stay as they are

B. A new if-branch in BalanceSheet for every split type

C. A new field on Expense for the weights

D. Every existing split gets a weights parameter

2. After any sequence of valid expenses, what is always true of a group's net balances?

A. Every balance is positive

B. They add up to zero

C. The payer's balance is the largest

D. They are all multiples of 100 paise

3. A PERCENT split of 10.50 with 33%, 33% and 34%. What are the shares?

A. 3.47, 3.46, 3.57

B. 3.46, 3.46, 3.57

C. 3.47, 3.47, 3.56

D. 3.50, 3.50, 3.50

4. How many payments does the greedy settle-up need at most for n members?

A. n - 1

B. n

C. n × (n - 1) / 2

D. log n

5. Is the greedy settle-up guaranteed to use the fewest possible transfers?

A. Yes, always

B. No; the true minimum can be lower, and finding it is NP-hard, so greedy is the practical choice

C. No; it often uses more than n - 1

D. Only when amounts are whole rupees

6. Why does the program break ties between equal creditors by name?

A. Splitwise does it that way

B. So the same input always prints the same payments, which a test or judge can compare

C. Names are cheaper to compare than amounts

D. To favour members who joined first

7. Which balance update is right for "asha pays 900 for herself, ben and chen, split equally"?

A. asha +900; asha, ben and chen -300 each

B. asha +600; ben and chen -300 each

C. Both of the above give the same net balances

D. asha -900; ben and chen +300 each

8. The app shows "ben owes asha" lines for every pair. Does the settle-up need those pairwise debts?

A. Yes; it settles each pair separately

B. No; it needs only each member's net balance

C. Yes; otherwise the totals are wrong

D. Only for groups larger than three

9. Two people add expenses to the same group at the same moment. What keeps the balances correct?

A. Nothing is needed; each expense is separate

B. Write the expense and its balance updates in one transaction, or recompute balances from the expenses

C. Round the amounts first

D. Let the second request overwrite the first

10. An expense names a participant who is not in the group. What should happen?

A. Refuse the whole expense before any balance changes

B. Apply it to the members who exist

C. Add the unknown name to the group

D. Charge the unknown share to the payer

ANSWER KEY · CHECKPOINT: EXPENSE SHARING

1. A That is the Strategy and the open/closed principle: the code that applies an expense depends only on the interface, so a new rule is a new class registered by name. Branching in BalanceSheet means every new rule edits tested code.

2. B An expense adds its amount to the payer and subtracts the same amount, as shares, from the participants: money is moved, never created. A total other than zero means a rounding or split bug, which makes it a cheap assertion.

3. A 1050 paise × 33 / 100 floors to 346, twice, and × 34 / 100 to 357: 1049 in all. The leftover paisa goes to the first member, giving 347, 346, 357. Without the leftover rule the shares would miss a paisa.

4. A Every payment brings at least one member to zero, and the last two members reach zero together, so at most n - 1 payments are made. Settling every pairwise debt separately can take up to n × (n - 1) / 2.

5. B When debts inside a subset of members cancel exactly, that subset can settle on its own with fewer transfers than greedy finds. Finding the minimum means searching subsets, which is NP-hard; greedy is short, fast and stays within n - 1.

6. B Without a fixed rule the choice falls to map iteration order, which for a HashMap is unspecified. A TreeMap walked with strict comparisons picks the earlier name every time, so the output is reproducible.

7. C Crediting 900 and debiting asha's own 300 nets to +600, the same as crediting only what the others owe. The exercise does the first because it is one rule for every case: payer +amount, every share -share, whether or not the payer took part.

8. B Who owes whom pairwise is useful to display, but settling only has to bring every net balance to zero, and nets are enough for that. That is also why it can use fewer transfers than paying each pairwise debt.

9. B Two read-modify-write updates of one balance can lose one of them. One transaction per expense, covering the expense row and the balances it moves, keeps them in step; recomputing from the expense rows avoids stored balances drifting at all.

10. A A half-applied expense breaks the zero-sum invariant and is hard to undo. Validate every name and the split's own rule first, then change all the balances, the same all-or-none rule as a multi-seat booking.
