Module 4 of 7

Vending machine

The State pattern with Idle, HasMoney and OutOfStock, greedy change, sold-out slots and restocking — then change from the coins actually in the box.

A vending machine is the textbook case for the State pattern: the same coin or button means something different depending on what the machine is doing. Instead of an if-chain on a status field in every method, each state is a class that answers every event and decides the next state.

You will code the state machine with greedy change, then add inventory: sales per slot, restocking that brings the machine back from OutOfStock, and a report. The checkpoint makes change from a coin box with a limited number of each coin — where greedy fails and a search is needed — and refunds the very coins inserted. Hidden cases run a machine that starts empty, rejected coins, a sold-out slot beside a full one, and change that cannot be made.

It is the practice half of the note Design a Vending Machine.

CONTENTS

1	Vending machine	12 min
✓	Checkpoint — Vending machine	35 min

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

LESSON 1 · 12 MIN

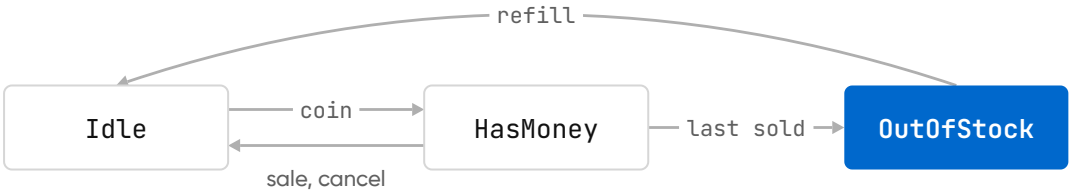
Vending machine

The textbook case for the **State pattern**: a coin means *start a sale* in one state, *add to the balance* in another and *give it back* in a third. The full design, with a sequence diagram and code in four languages, is the note Design a Vending Machine.

One class per state

- `VendingMachine` holds the **items** (code, name, price, stock), the **balance** and the **current state** — and no if-chain on a status.
- `interface State { insert, select, cancel }`, implemented by **Idle**, **HasMoney** and **OutOfStock**.
- Each state answers the event *and* decides the transition: `machine.move(next)`.
- States hold no data of their own, so one shared instance of each is enough.

One slot of chips through the vending machine's states




```
slot A1: chips, price 15, stock 0 balance 0
INSERT 5
→ sold out, returned 5
```

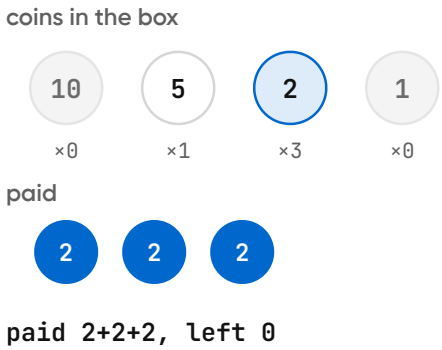
1. Three states, each a class answering every event. The arrows are the only ways between them, and each is taken by a state's own code calling machine.move.
2. Selecting in Idle starts nothing: Idle.select answers that coins come first, and the machine stays where it is.
3. A coin is the one event that leaves Idle: Idle.insert adds 10 to the balance and moves the machine to HasMoney.
4. The chips cost 15 and the balance is 10: HasMoney refuses the selection, keeps the money and stays in HasMoney.
5. In HasMoney a coin only raises the balance, to 20. The same INSERT event, a different class answering it.
6. HasMoney.select finds the price of 15 covered, drops the chips with change 5 and, as that was the last one, moves to OutOfStock rather than Idle.
7. OutOfStock cannot finish a sale, so it never starts one: OutOfStock.insert hands the coin straight back.

	INSERT	SELECT	CANCEL
Idle	add, → HasMoney	"insert coins first"	nothing to return
HasMoney	add	sell if paid, → Idle or OutOfStock	refund, → Idle
OutOfStock	return the coin	"sold out"	nothing to return

A coin the machine does not accept (only 1, 2, 5 and 10) is rejected before any state sees it.

Making change

Making change: greedy, and where a real coin box breaks it



1. The balance is 19 more than the price. Greedy change looks for the largest coin that still fits the amount left, and repeats until nothing is left.
2. 10 fits 19, and it is the largest coin, so it goes first, leaving 9.
3. 9 is left, so 10 no longer fits: the largest coin that does is 5, leaving 4.
4. 4 is left, so 10 and 5 no longer fit: the largest coin that does is 2, leaving 2.
5. A last 2 pays it off: 10 + 5 + 2 + 2 = 19 in 4 coins. For 10, 5, 2, 1 no shorter answer exists — this coin set is canonical.
6. Now the box holds one 5, three 2s and no 1s, and 6 is due. Greedy takes the 5 and is stuck with 1 to pay: there is no 1 left in the box.
7. Trying every count of each coin finds 2 + 2 + 2: three coins the box really holds. With limited coins, search the combinations — or refuse the sale before dispensing.

- **Greedy**: the largest coin that fits, repeat. For 10, 5, 2, 1 it gives the fewest coins.
- It is not optimal for every coin set, and a real box holds a **limited** number of each coin: then greedy can fail where an answer exists (6 from one 5 and three 2s is 2 + 2 + 2). The checkpoint makes you handle that.

Sold out, twice

- A **slot** with stock 0 refuses its selection; the balance stays.
- When **every** slot is empty, the machine is in OutOfStock and hands coins straight back. `REFILL` moves it back to Idle.

The traps

- `if (state == HAS_MONEY) … else if …` in every method: adding a state edits them all.
- Changing state from *outside* the states: transitions scattered over the machine are the if-chain again.

- Forgetting the balance on a refused selection: an underpaid or sold-out choice keeps the money in.

EXERCISE 1

The state machine

Model a vending machine with the **State pattern**: a `State` interface implemented by `Idle`, `HasMoney` and `OutOfStock`, and a machine that hands every event to its current state. The first line is the number of slots `n`; each of the next `n` lines is `<code> <name> <price> <stock>` (single words, whole numbers). The machine accepts coins of 1, 2, 5 and 10. Then events, one per line:

Command	Prints
`INSERT <coin>`	not 1, 2, 5 or 10: `rejected <coin>` (in any state). Idle → `balance <b>` and HasMoney; HasMoney → `balance <b>`; OutOfStock → `sold out, returned <coin>`
`SELECT <code>`	Idle → `insert coins first`; OutOfStock → `sold out`; HasMoney → `error: no slot <code>`, `<code> is sold out`, `need <k> more`, or `dispensed <name>, change <coins>` / `dispensed <name>, no change`
`CANCEL`	HasMoney → `returned <coins>` and Idle; otherwise `nothing to return`
`STATE`	`Idle`, `HasMoney` or `OutOfStock`
anything else	`error: unknown command <WORD>`

The machine starts in **Idle**, or in **OutOfStock** if every slot is empty. After a sale it moves to Idle — or to OutOfStock when that sale emptied the last slot. A refused selection keeps the balance. Change and refunds are paid greedily — the largest coin that fits, again and again — and printed largest first joined with `+` (`10+5+2+2`).

INPUT 1

```
2
A1 chips 15 1
A2 soda 12 2
SELECT A1
INSERT 10
SELECT A1
INSERT 10
SELECT A1
STATE
```

OUTPUT 1

```
insert coins first
balance 10
need 5 more
balance 20
dispensed chips, change 5
Idle
```

REFERENCE SOLUTION

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

public class Main {
    static class Item {
        final String code;
        final String name;
        final int price;
        int stock;
        int sold;

        Item(String code, String name, int price, int stock) { this.code = code; thi
    }

    interface State {
        String name();
        String insert(VendingMachine m, int coin);
        String select(VendingMachine m, String code);
        String cancel(VendingMachine m);
    }

    // States hold no data of their own, so one shared instance of each is enough.
    static final State IDLE = new Idle();
    static final State HAS_MONEY = new HasMoney();
    static final State OUT_OF_STOCK = new OutOfStock();

    static class Idle implements State {
        public String name() { return "Idle"; }

        public String insert(VendingMachine m, int coin) {
            m.balance += coin;
            m.move(HAS_MONEY);
            return "balance " + m.balance;
        }

        public String select(VendingMachine m, String code) { return "insert coins f

        public String cancel(VendingMachine m) { return "nothing to return"; }
    }

    static class HasMoney implements State {
        public String name() { return "HasMoney"; }

        public String insert(VendingMachine m, int coin) {
            m.balance += coin;
            return "balance " + m.balance;
        }

        public String select(VendingMachine m, String code) {
```

```
            Item item = m.items.get(code);
            if (item == null) return "error: no slot " + code;
            if (item.stock == 0) return code + " is sold out";
            if (m.balance < item.price) return "need " + (item.price - m.balance) +
            int change = m.balance - item.price;
            item.stock--;
            item.sold++;
            m.revenue += item.price;
            m.balance = 0;
            m.move(m.anyStock() ? IDLE : OUT_OF_STOCK);
            return "dispensed " + item.name + ", " + (change == 0 ? "no change" : "c
        }

        public String cancel(VendingMachine m) {
            String coins = coinsFor(m.balance);
            m.balance = 0;
            m.move(IDLE);
            return "returned " + coins;
        }
    }

    static class OutOfStock implements State {
        public String name() { return "OutOfStock"; }

        public String insert(VendingMachine m, int coin) { return "sold out, returne

        public String select(VendingMachine m, String code) { return "sold out"; }

        public String cancel(VendingMachine m) { return "nothing to return"; }
    }

    /** Greedy change: the largest coin that fits, again and again — the fewest coir
    static String coinsFor(int amount) {
        List<String> coins = new ArrayList<>();
        for (int c : new int[] {10, 5, 2, 1}) {
            while (amount >= c) {
                coins.add(String.valueOf(c));
                amount -= c;
            }
        }
        return String.join("+", coins);
    }

    static class VendingMachine {
        static final List<String> ACCEPTED = Arrays.asList("1", "2", "5", "10");

        final Map<String, Item> items = new TreeMap<>(); // sorted by code
        State state = IDLE;
        int balance = 0;
```

```
    int revenue = 0;

    void addItem(Item item) { items.put(item.code, item); }

    void start() { state = anyStock() ? IDLE : OUT_OF_STOCK; }

    void move(State next) { state = next; }

    boolean anyStock() {
        for (Item item : items.values()) if (item.stock > 0) return true;
        return false;
    }

    // Every event goes to the current state; the machine never asks which state
    String insert(String coin) {
        if (!ACCEPTED.contains(coin)) return "rejected " + coin;
        return state.insert(this, Integer.parseInt(coin));
    }

    String select(String code) { return state.select(this, code); }

    String cancel() { return state.cancel(this); }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int n = Integer.parseInt(br.readLine().trim());
    VendingMachine machine = new VendingMachine();
    for (int i = 0; i < n; i++) {
        String[] f = br.readLine().trim().split("\\s+");
        machine.addItem(new Item(f[0], f[1], Integer.parseInt(f[2]), Integer.par
    }
    machine.start();
    StringBuilder out = new StringBuilder();
    String line;
    while ((line = br.readLine()) ≠ null) {
        line = line.trim();
        if (line.isEmpty()) continue;
        String[] p = line.split("\\s+");
        String reply;
        switch (p[0]) {
            case "INSERT":
                reply = machine.insert(p[1]);
                break;
            case "SELECT":
                reply = machine.select(p[1]);
                break;
            case "CANCEL":
                reply = machine.cancel();
```

```
                    break;
                case "STATE":
                    reply = machine.state.name();
                    break;
                default:
                    reply = "error: unknown command " + p[0];
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

EXERCISE 2

Inventory and restocking

Extend the state machine with inventory. The first line is the number of slots `n`; each of the next `n` lines is `<code> <name> <price> <stock>` (single words, whole numbers). The machine accepts coins of 1, 2, 5 and 10. Every event of the first program works as before:

Command	Prints
`INSERT <coin>`	not 1, 2, 5 or 10: `rejected <coin>` (in any state). Idle → `balance <b>` and HasMoney; HasMoney → `balance <b>`; OutOfStock → `sold out, returned <coin>`
`SELECT <code>`	Idle → `insert coins first`; OutOfStock → `sold out`; HasMoney → `error: no slot <code>`, `<code> is sold out`, `need <k> more`, or `dispensed <name>, change <coins>` / `dispensed <name>, no change`
`CANCEL`	HasMoney → `returned <coins>` and Idle; otherwise `nothing to return`
`STATE`	`Idle`, `HasMoney` or `OutOfStock`

and three maintenance commands are added — they work in **any** state:

Command	Prints
`REFILL <code> <count>`	`<code> stock <new stock>`, or `error: no slot <code>`. In OutOfStock, the machine moves to Idle once any slot has stock again (a refill of 0 keeps it OutOfStock); in other states the state is kept.
`REPORT`	one line per slot, in code order: `<code> <name>: stock <s>, sold <k>`
`TOTAL`	`revenue <sum of the prices of every item sold>`
anything else	`error: unknown command <WORD>`

The machine starts in **Idle**, or in **OutOfStock** if every slot is empty. After a sale it moves to Idle — or to OutOfStock when that sale emptied the last slot. A refused selection keeps the balance. Change and refunds are greedy, printed largest first joined with `+`.

INPUT 1

```
2
B2 water 10 1
A1 chips 15 0
INSERT 10
SELECT B2
STATE
REFILL A1 2
STATE
INSERT 10
INSERT 5
SELECT A1
REPORT
TOTAL
```

OUTPUT 1

```
balance 10
dispensed water, no change
OutOfStock
A1 stock 2
Idle
balance 10
balance 15
dispensed chips, no change
A1 chips: stock 1, sold 1
B2 water: stock 0, sold 1
revenue 25
```

REFERENCE SOLUTION

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

public class Main {
    static class Item {
        final String code;
        final String name;
        final int price;
        int stock;
        int sold;

        Item(String code, String name, int price, int stock) { this.code = code; thi
    }

    interface State {
        String name();
        String insert(VendingMachine m, int coin);
        String select(VendingMachine m, String code);
        String cancel(VendingMachine m);
    }

    // States hold no data of their own, so one shared instance of each is enough.
    static final State IDLE = new Idle();
    static final State HAS_MONEY = new HasMoney();
    static final State OUT_OF_STOCK = new OutOfStock();

    static class Idle implements State {
        public String name() { return "Idle"; }

        public String insert(VendingMachine m, int coin) {
            m.balance += coin;
            m.move(HAS_MONEY);
            return "balance " + m.balance;
        }

        public String select(VendingMachine m, String code) { return "insert coins f

        public String cancel(VendingMachine m) { return "nothing to return"; }
    }

    static class HasMoney implements State {
        public String name() { return "HasMoney"; }

        public String insert(VendingMachine m, int coin) {
            m.balance += coin;
            return "balance " + m.balance;
        }

        public String select(VendingMachine m, String code) {
```

```
        Item item = m.items.get(code);
        if (item == null) return "error: no slot " + code;
        if (item.stock == 0) return code + " is sold out";
        if (m.balance < item.price) return "need " + (item.price - m.balance) +
        int change = m.balance - item.price;
        item.stock--;
        item.sold++;
        m.revenue += item.price;
        m.balance = 0;
        m.move(m.anyStock() ? IDLE : OUT_OF_STOCK);
        return "dispensed " + item.name + ", " + (change == 0 ? "no change" : "c
    }

    public String cancel(VendingMachine m) {
        String coins = coinsFor(m.balance);
        m.balance = 0;
        m.move(IDLE);
        return "returned " + coins;
    }
}

static class OutOfStock implements State {
    public String name() { return "OutOfStock"; }

    public String insert(VendingMachine m, int coin) { return "sold out, returne

    public String select(VendingMachine m, String code) { return "sold out"; }

    public String cancel(VendingMachine m) { return "nothing to return"; }
}

/** Greedy change: the largest coin that fits, again and again — the fewest coir
static String coinsFor(int amount) {
    List<String> coins = new ArrayList<>();
    for (int c : new int[] {10, 5, 2, 1}) {
        while (amount >= c) {
            coins.add(String.valueOf(c));
            amount -= c;
        }
    }
    return String.join("+", coins);
}

static class VendingMachine {
    static final List<String> ACCEPTED = Arrays.asList("1", "2", "5", "10");

    final Map<String, Item> items = new TreeMap<>(); // sorted by code
    State state = IDLE;
    int balance = 0;
```

```
    int revenue = 0;

    void addItem(Item item) { items.put(item.code, item); }

    void start() { state = anyStock() ? IDLE : OUT_OF_STOCK; }

    void move(State next) { state = next; }

    boolean anyStock() {
        for (Item item : items.values()) if (item.stock > 0) return true;
        return false;
    }

    // Every event goes to the current state; the machine never asks which state
    String insert(String coin) {
        if (!ACCEPTED.contains(coin)) return "rejected " + coin;
        return state.insert(this, Integer.parseInt(coin));
    }

    String select(String code) { return state.select(this, code); }

    String cancel() { return state.cancel(this); }

    /** Restocking is maintenance, not a customer event: any state, and it can e
    String refill(String code, int count) {
        Item item = items.get(code);
        if (item == null) return "error: no slot " + code;
        item.stock += count;
        if (state == OUT_OF_STOCK && anyStock()) move(IDLE);
        return code + " stock " + item.stock;
    }

    String report() {
        StringBuilder sb = new StringBuilder();
        for (Item item : items.values()) {
            if (sb.length() > 0) sb.append('\n');
            sb.append(item.code).append(' ').append(item.name).append(": stock '
        }
        return sb.toString();
    }
}

public static void main(String[] args) throws IOException {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    int n = Integer.parseInt(br.readLine().trim());
    VendingMachine machine = new VendingMachine();
    for (int i = 0; i < n; i++) {
        String[] f = br.readLine().trim().split("\\s+");
        machine.addItem(new Item(f[0], f[1], Integer.parseInt(f[2]), Integer.par
```

```
        }
        machine.start();
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            String reply;
            switch (p[0]) {
                case "INSERT":
                    reply = machine.insert(p[1]);
                    break;
                case "SELECT":
                    reply = machine.select(p[1]);
                    break;
                case "CANCEL":
                    reply = machine.cancel();
                    break;
                case "STATE":
                    reply = machine.state.name();
                    break;
                case "REFILL":
                    reply = machine.refill(p[1], Integer.parseInt(p[2]));
                    break;
                case "REPORT":
                    reply = machine.report();
                    break;
                case "TOTAL":
                    reply = "revenue " + machine.revenue;
                    break;
                default:
                    reply = "error: unknown command " + p[0];
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

QUIZ

1. Why model a vending machine with the State pattern instead of a `status` field and if-else?

A. It runs faster

B. Each state answers every event in one class, so a new state is a new class instead of a new branch in every method

C. Java has no switch on strings

D. It removes the need for a balance

2. The machine is OutOfStock and a customer inserts a 10. What should happen?

A. Balance 10, move to HasMoney

B. The coin is returned at once

C. The coin is kept

D. The machine crashes

3. Greedy change for 19 with coins 10, 5, 2, 1 gives:

A. 10+5+2+2

B. 10+5+4

C. 10+9

D. 5+5+5+2+2

4. A customer has 10 in and selects an item costing 15. What does HasMoney do?

A. Dispense and owe 5

B. Refuse with "need 5 more" and keep the balance and the state

C. Refund 10 and go Idle

D. Move to OutOfStock

ANSWER KEY · VENDING MACHINE

1. B With a status field, every event method grows a branch per status, and adding Maintenance edits all of them. With State, the behaviour for one state is together and transitions are explicit.

2. B OutOfStock cannot finish a sale, so it must not start one: it returns the coin. That is exactly the kind of rule that lives in the state's own `insert`.

3. A Take 10 (9 left), 5 (4 left), 2 (2 left), 2 (0 left): four coins, the fewest possible for this canonical set.

4. B An underpaid selection is refused without losing the money: the balance stays and the machine stays in HasMoney, waiting for more coins or a cancel.

CHECKPOINT · 35 MIN

Checkpoint — Vending machine

This checkpoint covers the module: the State pattern and its transitions, coins rejected before any state sees them, sold-out slots versus a sold-out machine, restocking, and making change.

How it works. Ten questions and one program. You need 70% on the questions and the program accepted to clear the module.

The program is the hardest in this track so far. The machine now pays change from a coin box holding a limited number of each coin, and inserted coins wait in escrow until a sale. Before you code, decide:

- Which state owns the "can I pay the change?" check, and what happens to the escrow if the answer is no?
- How do you find the fewest coins when greedy can fail? (The amounts are small — a search over counts is fine.)
- What does a cancel return: a greedy refund, or the coins the customer actually inserted?

Hidden cases make change from the customer's own coins, refuse a sale for want of change, and empty the machine.

EXERCISE

Change from a real coin box

A real machine pays change from the coins it holds. The first line is the number of slots `n`; each of the next `n` lines is `<code> <name> <price> <stock>` (single words, whole numbers). The machine accepts coins of 1, 2, 5 and 10. The next line is the coin box: how many coins of 10, 5, 2 and 1 it holds, in that order. Then events.

Inserted coins are held **apart** (in escrow) until a sale: `CANCEL` returns those very coins in the order inserted, joined with `+`. On a paid selection the inserted coins go into the box, and the change is paid with **the fewest coins the box can make it with** — among equally few, the one with the most 10s, then 5s, then 2s. If no combination of the box's coins makes the change, the sale is refused: the inserted coins come back out of the box and are returned, the item stays, and the machine is Idle.

Command	Prints
`INSERT <coin>`	`rejected <coin>` if not 1, 2, 5 or 10; Idle/HasMoney → `balance <b>`; OutOfStock → `sold out, returned <coin>`
`SELECT <code>`	Idle → `insert coins first`; OutOfStock → `sold out`; HasMoney → `error: no slot <code>`, `<code> is sold out`, `need <k> more`, `no change for <amount>, returned <coins inserted>`, or `dispensed <name>, change <coins, largest first>` / `dispensed <name>, no change`
`CANCEL`	HasMoney → `returned <coins inserted>`; otherwise `nothing to return`
`STATE`	`Idle`, `HasMoney` or `OutOfStock`
`COINS`	`box: 10x<a> 5x<b> 2x<c> 1x<d>`
anything else	`error: unknown command <WORD>`

The machine starts Idle, or OutOfStock if every slot is empty; a sale that empties the last slot moves it to OutOfStock.

INPUT 1

```
1
A1 chips 4 5
0 1 3 0
INSERT 10
SELECT A1
COINS
```

OUTPUT 1

```
balance 10
dispensed chips, change 2+2+2
box: 10x1 5x1 2x0 1x0
```

REFERENCE SOLUTION

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

public class Main {
    static final int[] DENOMS = {10, 5, 2, 1};

    static class Item {
        final String code;
        final String name;
        final int price;
        int stock;

        Item(String code, String name, int price, int stock) { this.code = code; thi
    }

    interface State {
        String name();
        String insert(VendingMachine m, int coin);
        String select(VendingMachine m, String code);
        String cancel(VendingMachine m);
    }

    static final State IDLE = new Idle();
    static final State HAS_MONEY = new HasMoney();
    static final State OUT_OF_STOCK = new OutOfStock();

    static class Idle implements State {
        public String name() { return "Idle"; }

        public String insert(VendingMachine m, int coin) {
            m.escrow.add(coin);
            m.move(HAS_MONEY);
            return "balance " + m.balance();
        }

        public String select(VendingMachine m, String code) { return "insert coins f

        public String cancel(VendingMachine m) { return "nothing to return"; }
    }

    static class HasMoney implements State {
        public String name() { return "HasMoney"; }

        public String insert(VendingMachine m, int coin) {
            m.escrow.add(coin);
            return "balance " + m.balance();
        }

        public String select(VendingMachine m, String code) {
```

```
            Item item = m.items.get(code);
            if (item == null) return "error: no slot " + code;
            if (item.stock == 0) return code + " is sold out";
            int balance = m.balance();
            if (balance < item.price) return "need " + (item.price - balance) + " mo
            int change = balance - item.price;
            m.box.add(m.escrow); // the customer's coins can be part of the change
            int[] pay = m.box.makeChange(change);
            if (pay == null) {
                m.box.remove(m.escrow);
                String back = joined(m.escrow);
                m.escrow.clear();
                m.move(IDLE);
                return "no change for " + change + ", returned " + back;
            }
            m.box.take(pay);
            m.escrow.clear();
            item.stock--;
            m.move(m.anyStock() ? IDLE : OUT_OF_STOCK);
            return "dispensed " + item.name + ", " + (change == 0 ? "no change" : "c
        }

        public String cancel(VendingMachine m) {
            String back = joined(m.escrow); // the very coins inserted, held apart u
            m.escrow.clear();
            m.move(IDLE);
            return "returned " + back;
        }
    }

    static class OutOfStock implements State {
        public String name() { return "OutOfStock"; }

        public String insert(VendingMachine m, int coin) { return "sold out, returne

        public String select(VendingMachine m, String code) { return "sold out"; }

        public String cancel(VendingMachine m) { return "nothing to return"; }
    }

    static String joined(List<Integer> coins) {
        StringBuilder sb = new StringBuilder();
        for (int c : coins) sb.append(sb.length() == 0 ? "" : "+").append(c);
        return sb.toString();
    }

    /** The coins the machine holds, counted per denomination (10, 5, 2, 1). */
    static class CoinBox {
        final int[] count = new int[DENOMS.length];
```

```
    void add(List<Integer> coins) { for (int c : coins) count[index(c)]++; }

    void remove(List<Integer> coins) { for (int c : coins) count[index(c)]--; }

    void take(int[] pay) { for (int i = 0; i < DENOMS.length; i++) count[i] -= p

    /**
     * The fewest coins the box can pay the amount with, or null. Greedy is not
     * coins run out (6 from one 5 and three 2s is 2+2+2), so try every count of
     * most first: among equally short answers, the one with the most large coin
     */
    int[] makeChange(int amount) {
        int[] best = null;
        int bestCoins = Integer.MAX_VALUE;
        for (int a = Math.min(count[0], amount / 10); a >= 0; a--) {
            int r1 = amount - 10 * a;
            for (int b = Math.min(count[1], r1 / 5); b >= 0; b--) {
                int r2 = r1 - 5 * b;
                for (int c = Math.min(count[2], r2 / 2); c >= 0; c--) {
                    int d = r2 - 2 * c;
                    if (d > count[3]) continue;
                    int coins = a + b + c + d;
                    if (coins < bestCoins) {
                        bestCoins = coins;
                        best = new int[] {a, b, c, d};
                    }
                }
            }
        }
        return best;
    }

    static String describe(int[] pay) {
        List<String> coins = new ArrayList<>();
        for (int i = 0; i < DENOMS.length; i++) for (int k = 0; k < pay[i]; k++)
        return String.join("+", coins);
    }

    public String toString() {
        StringBuilder sb = new StringBuilder("box:");
        for (int i = 0; i < DENOMS.length; i++) sb.append(' ').append(DENOMS[i])
        return sb.toString();
    }

    private static int index(int coin) {
        for (int i = 0; i < DENOMS.length; i++) if (DENOMS[i] == coin) return i;
        throw new IllegalArgumentException("coin " + coin);
    }
```

```
    }

    static class VendingMachine {
        static final List<String> ACCEPTED = Arrays.asList("1", "2", "5", "10");

        final Map<String, Item> items = new TreeMap<>();
        final CoinBox box = new CoinBox();
        final List<Integer> escrow = new ArrayList<>(); // inserted, not yet the mac
        State state = IDLE;

        int balance() {
            int sum = 0;
            for (int c : escrow) sum += c;
            return sum;
        }

        void start() {
            state = anyStock() ? IDLE : OUT_OF_STOCK;
        }

        void move(State next) { state = next; }

        boolean anyStock() {
            for (Item item : items.values()) if (item.stock > 0) return true;
            return false;
        }

        String insert(String coin) {
            if (!ACCEPTED.contains(coin)) return "rejected " + coin;
            return state.insert(this, Integer.parseInt(coin));
        }

        String select(String code) { return state.select(this, code); }

        String cancel() { return state.cancel(this); }
    }

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(br.readLine().trim());
        VendingMachine machine = new VendingMachine();
        for (int i = 0; i < n; i++) {
            String[] f = br.readLine().trim().split("\\s+");
            Item item = new Item(f[0], f[1], Integer.parseInt(f[2]), Integer.parseIr
            machine.items.put(item.code, item);
        }
        String[] counts = br.readLine().trim().split("\\s+");
        for (int i = 0; i < DENOMS.length; i++) machine.box.count[i] = Integer.parse
        machine.start();
```

```
        StringBuilder out = new StringBuilder();
        String line;
        while ((line = br.readLine()) != null) {
            line = line.trim();
            if (line.isEmpty()) continue;
            String[] p = line.split("\\s+");
            String reply;
            switch (p[0]) {
                case "INSERT":
                    reply = machine.insert(p[1]);
                    break;
                case "SELECT":
                    reply = machine.select(p[1]);
                    break;
                case "CANCEL":
                    reply = machine.cancel();
                    break;
                case "STATE":
                    reply = machine.state.name();
                    break;
                case "COINS":
                    reply = machine.box.toString();
                    break;
                default:
                    reply = "error: unknown command " + p[0];
            }
            out.append(reply).append('\n');
        }
        System.out.print(out);
    }
}
```

CHECKPOINT QUESTIONS

1. In the State pattern, who decides the next state after an event?

A. The caller (main)

B. The current state object, by calling machine.move(next)

C. A global switch statement

D. The Item

2. Why can the states be shared singletons (`IDLE`, `HAS_MONEY`)?

A. Java requires it

B. They hold no data of their own — the balance and items live on the machine they are passed

C. To make the machine thread-safe

D. Because there is only one machine

3. The box holds one 5 and three 2s, and the change due is 6. What does greedy-from-the-box do, and what is right?

A. Both pay 5+1

B. Greedy takes the 5 and is stuck needing 1; the right answer is 2+2+2

C. Both pay 2+2+2

D. Neither can pay

4. Why hold inserted coins in escrow instead of dropping them straight into the box?

A. To count them faster

B. So a cancel or a refused sale returns the very coins the customer put in

C. Escrow is required by law

D. To avoid the State pattern

5. A sale empties the last item in the machine. Which state comes next?

A. Idle

B. HasMoney

C. OutOfStock

D. Dispensing forever

6. A slot is sold out but others have stock. A customer with money selects the sold-out slot. What happens?

A. The machine moves to OutOfStock

B. The selection is refused and the balance stays in HasMoney

C. The money is refunded

D. Another item is dispensed

7. You need a Maintenance state in which every customer event is refused. What changes?

A. Every existing state class

B. One new class implementing State, plus the transitions into and out of it

C. The Item class

D. Nothing can be added

8. Where should the check "is this a coin the machine accepts?" live?

A. In every state's insert

B. Once, in the machine, before the event reaches any state

C. In main only

D. In the CoinBox's toString

9. Greedy change is guaranteed optimal for which coin set?

A. 4, 3, 1

B. 10, 5, 2, 1

C. Any set containing 1

D. Any set at all

10. Is Dispensing needed as its own state in a single-threaded program like this one?

A. Yes, always

B. Not here — the sale completes within one call; in a real machine, events can arrive while the motor turns, so it is needed there

C. No state is needed

D. Only for refunds

ANSWER KEY · CHECKPOINT — VENDING MACHINE

1. B Each state knows where its events lead, so transitions live with the behaviour. A machine that sets its own state from outside the states is the if-chain again.

2. B A stateless state object can serve every machine. If a state needed its own data (a timer, say), each machine would create its own instance instead.

3. B Greedy is optimal for unlimited canonical coins, not for a limited box. Searching the combinations finds 2+2+2.

4. B Until a sale completes, the coins are still the customer's. Keeping them apart makes a refund exact and keeps the box's count true.

5. C With nothing left to sell, the machine moves to OutOfStock, where coins are returned at once. A refill moves it back to Idle.

6. B A sold-out slot is not a sold-out machine. HasMoney refuses that one selection and waits for another choice or a cancel.

7. B That is the open-closed payoff of the State pattern: the new behaviour is a new class, and the existing states change only where they transition into it.

8. B A foreign coin is rejected whatever the state, so the check belongs before the delegation — one place instead of three copies.

9. B 10, 5, 2, 1 is a canonical system, where greedy always gives the fewest coins. For 4, 3, 1, greedy pays 6 as 4+1+1 although 3+3 is shorter.

10. B A state exists to answer events differently. If no event can arrive during dispensing, the state would never answer one; on real hardware, it must refuse coins mid-sale.