Vending Machine (2D Arrays)

You'll write the methods from the VendingMachine FRQ in real code and test them. Create a new project in IntelliJ and copy all the .java files to the src folder. You can compile and run Tester. Your job is to make all the tests pass. First write part (A) of the FRQ. Then an easier part (B) where the snack machine is known to be full. Finally the original part (B) for any snack machine.

You can run VendingMachineApp to interact with the program

As a recap, VendingMachine is a 2D array of Snacks. Snacks have names and prices. You can view the Snack class if you would like. You are only modifying VendingMachine.java

Files

File	What it is
`Snack.java`	Done. A snack has a name and a price.
`VendingMachine.java`	**You edit this.** Holds `private Snack[][] slots`. An empty slot is `null`.
`VendingMachineApp.java`	A text vending machine to try your methods by hand.
`Tester.java`	Run this to check your work.

`Tester` prints a line for each test, like

```
FAIL: FRQ example
    restock(0, crackers) expected true received false
```

A snack is shown with the slot it started in, like `mints $0.75 [0][3]`, so you can tell two snacks with the same name apart. Fix the first failure you see, then run it again.

Part (a): `restock(int row, Snack s)`

Put `s` in the **leftmost empty** slot of row `row` and return `true`. If the row has no empty slots, change nothing and return `false`.

Part (b1): `buyCheapestFull()` (warm-up)

For this method, assume **every slot holds a snack**: there are no `null`s.

Remove the cheapest snack from the machine (set its slot to `null`) and return it.

If more than one snack has the lowest price, take the **first** one you find when checking slots in this order: the last column from the bottom row up to the top,

then the next-to-last column from the bottom up, and so on, ending with the first column. For a 3 x 4 machine:

	col 0	col 1	col 2	col 3
row 0	12	9	6	3
row 1	11	8	5	2
row 2	10	7	4	1

Run `Tester` and get all of Part (b1) passing before you go on.

Part (b2): `buyCheapest()`

Same as (b1), except now **any slot may be empty**. Skip `null` slots: calling `getPrice()` on `null` crashes. If the machine has no snacks at all, change nothing and return `null`.

Think about what your (b1) code used as its starting "cheapest so far." Why doesn't that work anymore?