Exam 1

- #1a is about correctness – be sure to account for everything listed, and to both identify (comments) and appropriately check (code) each thing
  - The cash register cannot contain a negative number of bills/coins of any denomination, and denominations can only be positive numbers.
    - class invariant: `denominations_[i] > 0`, `counts_[i] > 0`
      - state in comments by the instance variable declarations
      - check using assertions at the end of every constructor/method that can change those values – the constructor, `add`, and `dispense`
  - There can only be one slot in the drawer for a given denomination.
    - the denominations are initialized in the constructor, so this is a precondition for the constructor's parameter
      - state that the denominations must be unique in the constructor's comment
      - check at the beginning of the constructor body – throw an `IllegalArgumentException` if violated

Exam 1

- #1 is about correctness and robustness – be sure to account for everything listed, and to both identify (comments) and appropriately check (code) each thing
  - Counts and amounts cannot be negative.
    - preconditions: `count >= 0` for `add`, `dispense`; `amount >= 0` for `makeChange`
      - state in the comments for those methods
      - check at the beginning of the method bodies – throw an `IllegalArgumentException` if violated

Exam 1

- #1 is about correctness and robustness – be sure to account for everything listed, and to both identify (comments) and appropriately check (code) each thing
  - The cash register cannot dispense more of a given denomination than it holds, and it may be unable to make change if it doesn't have enough of certain denominations.
    - that it cannot dispense more than it holds is a precondition for dispense: `count <= getCount(denom)`
      - state in the comments for `dispense`
      - check at the beginning of the method body – throw an IllegalArgumentException if violated
    - not being able to make change is an error, but not a precondition – there's no way for the caller to know if there's the right amount of change
      - state in the comments for `makeChange`
      - if in the process of making change it is found that there isn't enough change, throw an exception – this isn't an IllegalArgumentException and it is fine to just throw an `Exception` rather than a more specific type

Exam 1

- #1b is about robustness – be sure to check for and appropriately handle what can go wrong
  - a number < 0 is entered for `denom`
    - check with an `if` statement – print an error message and go to the next iteration of the loop (`continue`)
      - the clerk should be given another chance to enter a valid value
  - a number < 0 is entered for `count`
    - check with an if statement – print an error message and go to the next iteration of the loop (continue)
      - the clerk should be given another chance to enter a valid value
  - a non-integer is entered for `denom` or `count`
    - `Scanner` will throw an `InputMismatchException` – put a try-catch block around the body of the loop so that the loop continues with the next iteration and print an informative error message for the user in the catch block
      - the clerk should be given another chance to enter a valid value
  - `makeChange` might not be able to make change
    - put a try-catch block around the `makeChange` call to catch the exception and print an informative error message
  - not enough is paid (`paid < total`)
    - add an `else` and print an informative error message

Exam 1

- #3 – test cases are about testing correct behavior
  - e.g. don’t need to test if violated preconditions throw an exception