This is intended to be a coding exercise done at home, at a time convenient to you, using the tools and libraries that you're comfortable working with. We will be looking for clean, readable, correct code over code highly optimized for performance. There is no limit to the amount of time you spend on it, but it is feasible to build a complete, working solution in a couple of hours.

Problem Constraints

Before looking at the problem itself, please read the following constraints:

- The solution should be implemented using Java (compatible with Java 8)
- It should go without saying that all code submitted should be created by you for the purpose of this exercise and not taken from other projects or other sources
- Please provide the solution in the form of a zip file containing those files you would normally check in to version control (such as git)
    - The solution should not contain IDE specific files (i.e. IntelliJ projects), but should include a build file for the tool of your choice for build and dependency management
- You are free to use all of the standard Java framework as well as any general purpose open source framework that you wish - please ensure that any external dependencies are included in the build files that you provide
- The solution is intended to be standalone, with no external runtime dependencies. You do not have to worry about multiple processes, and can keep all state in memory. In addition, you can disregard all security concerns and there are no requirements for monitoring or diagnostics above the operations explicitly described below.
- Your implementation should be thread safe and each individual operation should be atomic, i.e. either fail and leave the state untouched, or succeed and finish in a completely updated state. No support for cross operation transactions is required.
- No main() or UI is required, but you do need to demonstrate that the solution is working correctly
- While raw performance is not the priority for this exercise, please document any tradeoffs that you have made between performance, readability, maintainability etc in JavaDoc
- Overall, the solution you submit should meet your standard for "production ready", but we will not be looking for 100% JavaDoc or 100% test coverage as a goal in itself

Coding Challenge - Vending Machine

Your challenge is to design and implement a class representing a basic vending machine capable of keeping track of the number of items of each type currently in the machine, the amount of change currently in the machine for each type of coin, and to return correct change given a product selection and a set of coins submitted. The solution is expected to be in the form of two interfaces (see below) and a class implementing these interfaces, along with any helpers etc that you may wish to create.

Each vending machine instance has a fixed number of product slots, supports a fixed number of coin types (both set at instantiation time), but can have price and quantity for each product slot adjusted as well we the amount of change available - imagine an operator collecting money from the machine, filling it up with new items and potentially changing which products are being sold.

The vending machine also supports price display and actual purchases of products using coins provided by a user. This operation is expected to work the way a vending machine would, i.e. buying a product decreases the inventory for that product by one, and the amount of change in the machine is updated.

Please focus on the vending machine itself rather than a rich domain around products. It is perfectly fine to represent a product with simply a string or number. Also note that the spec is intentionally loose and all method signatures or names are for illustration purposes only, please select method names, method signatures etc to your liking as long as the implementation supports the behavior outlined.

Constructing a vending machine instance

It should be possible to instantiate your vending machine class with a set of product slots, and a set of supported coin types. While the number of available products per slot, product price, and amount of change available can all change (see below), the set of product slots and supported coin types should be seen as immutable for any given instance of the vending machine. An example constructor invocation may be

```
new VendingMachine(10, Arrays.asList(0.10, 0.20, 0.50, 1.0))
```

… to support 10 product slots (implicitly identified as 0..9), and coins worth 10p, 20p, 50p and £1, but you are free to choose the types as you see fit.

Performing maintenance on the vending machine

Design and implement a vending machine interface to support the following maintenance operations:

- Set/get the quantity of items for a given product slot
- Set/get the price per item for a given product slot
- Set/get the amount of coins available for a given type of coin

The implementation should be thread safe and throw an appropriate exception for operations that do not make sense, e.g. `IllegalStateException` if setting the quantity of items for a product slot with no price specified, or `IllegalArgumentException` if trying to set the price for a product slot that the machine does not have.

Consumer use of the vending machine

Design and implement an interface representing operations that can be performed by users of the vending machine:

- Get the price per item for a given product slot
- Buy the product for a given product slot. This should be one operation that accepts a product slot identifier, a collection of coins, and returns another collection of coins representing the change given if successful, e.g:
  `buyProduct(1, Arrays.asList(1.0, 1.0))`
  might return an array of {0.5, 0.2, 0.1} if the product costs £1.30

The implementation should throw `IllegalStateException` if the current state of the machine does not support the operation (e.g. the item is sold out, the item doesn't have a price), and should throw `IllegalArgumentException` if the parameters are invalid (non existing product slot referenced, not enough money provided).

Performance is a secondary concern, but the implementation should always allow for a purchase as long as exact change in any form can be returned. Please briefly document the strategy that you've chosen for providing the "best" set of coins returned any any trade offs you have done.