JAVA Challenge

Please implement the challenge according to specification and return it as .tar.gz package. The implementation should be done in Java 8. You are free to use external libraries in your solution.

This challenge is for private use only, we ask you to not publish your solution or the challenge.

Specification:

We have a new client whose main business is ready-made lunches, including sandwiches and fried eggs. All lunches are made from multiple components (eggs, bread, mayonnaise, various cold cuts, etc.), which may themselves be intermediate products that are made of components (such as mayonnaise) or basic ingredients that do not have further components (e.g. oil, yeast). This is described for each sandwich type as a bill of materials (BOM), supplied by the client.

Each bill of materials is a list of product-component relations, with a multiplier describing how many units of the component is needed for making the product.

Bill of materials list forms a tree structure, where the top-level product (such as sandwich or fried eggs) is the root and components are either intermediate items (like mayo) or leaves (such as yeast).

All bills of materials are supplied in a CSV file. However, you don't have to implement any file I/O functionality. Instead, there is a CSV-reading utility that provides a simple API to its contents.

Your solution should reside under package `com.relexsolutions.javaassignment.sol` . Besides the solution and the tests, there should be no need to touch other parts of the project. Provided classes such as `DataProvider` should be left as is - unless there's some justifiable reason for changes.

Note that the example API calls and their return values are described in pseudo-code. You may change types supplied to and returned from the provided methods and interfaces as you see fit. Supplied unit tests can also be changed as required as long as the intent remains the same. Thus, everything within the project may be changed if you deem it necessary.

Part1:

The client is interested in automating the ordering of basic ingredients based on the needs of different sandwiches. To facilitate this, we need an intermediate structure to illustrate how each type of sandwich is eventually deconstructed to its basic ingredients.

Given an input with the following structure:

code	**component_code**	**start_date**	**end_date**	**multiplier**
CLUB_SANDWICH	BREAD			1
BREAD	FLOUR			1
BREAD	YEAST			1
CLUB_SANDWICH	MAYO			1
MAYO	EGG			1
EGG	EGG_WHITE			1
EGG	EGG_YOLK			1
MAYO	OIL			1
MAYO	MUSTARD			1
MUSTARD	GARLIC_POWDER			1
GARLIC_POWDER	GARLIC			1
CLUB_SANDWICH	BACON			1
FRIED_EGGS	EGG			1
FRIED_EGGS	OIL			1

Construct a tree structure that implements the given API as follows: `Set<String> getCodes`
for a given product code, anywhere in the tree, it returns the values of ancestor products in an unordered collection, like Set. `Set<String> getCodesForComponents(String produ`
for a given product code, anywhere in the tree, it returns the values of child products in an unordered collection, like Set. `List<Map> getComponents(String productCode)`
for a given product code, anywhere in the tree, it returns every component (children) for all levels. Returned value should be an ordered list (you may define the order as you see fit) of key-value pairs (Maps) containing following information: product code, depth in the current tree, multiplier.

```
getCodesForProducts("GARLIC_POWDER") => Set("MUSTARD", "MAYO", "CL
getCodesForComponents("CLUB_SANDWICH") => Set("BREAD", "MAYO", "BA

getComponents("CLUB_SANDWICH") => [
    {code: "BREAD", depth: 1, multiplier 1},
    {code: "FLOUR", depth: 2, multiplier 1},
    {code: "YEAST", depth: 2, multiplier 1},
    {code: "MAYO", depth: 1, multiplier 1},
    {code: "EGG", depth: 2, multiplier 1},
    {code: "EGG_WHITE", depth: 3, multiplier 1},
    {code: "EGG_YOLK", depth: 3, multiplier 1},
    {code: "OIL", depth: 2, multiplier 1},
    {code: "MUSTARD", depth: 2, multiplier 1},
    {code: "GARLIC_POWDER", depth: 4, multiplier 1},
    {code: "GARLIC", depth: 3, multiplier 1},
    {code: "BACON", depth: 1, multiplier 1},
]

getComponents("GARLIC_POWDER") => [
    {code: "GARLIC", depth: 1, multiplier 1},
]

getComponents("GARLIC") => []

getComponents("FRIED_EGGS") = [
    {code: "EGG", depth: 2, multiplier 1},
    {code: "EGG_WHITE", depth: 3, multiplier 1},
    {code: "EGG_YOLK", depth: 3, multiplier 1},
    {code: "OIL", depth: 2, multiplier 1}
]
```

NB! The data is supplied by client's ERP and its structural correctness is not guaranteed. However, you can assume that the data is syntactically valid. In practice this means that you can assume you are always getting Java objects though `DataProvider` *APIs (instead e.g. of having to catch exceptions), but the objects' properties may be anything*

Part2:

Because of supplier restrictions and limited ingredient availability, some basic ingredients may not be available (i.e. active) always. This appears in the bill of materials so that ingredients may have a start date and an end date defined. When an ingredient has a start and end date defined and its multiplier set to zero, it’s considered inactive for that period. If an ingredient has only start date defined it means that it's active from that date to eternity. In similar fashion having only end date defined means that the ingredient is active from beginning of time up to the given date.

In most cases, the client is able to substitute an inactive ingredient with other ingredients. Thus there may be other, similar ingredients listed as active during that time. If you have a an ingredient replacing another for any time range (e.g. HYPER_YEAST replaces SUPER_YEAST on 2016-12-24) you also need to explicitly set the replaced ingredient inactive (zero multiplier) for the given period or otherwise the both ingredients would be considered active for the time range.

Active/inactive ingredients appear in the data as follows:

code	**component_code**	**start_date**	**end_date**	**multiplier**
CLUB_SANDWICH	BREAD			1
BREAD	FLOUR			1
BREAD	YEAST			1
BREAD	YEAST	2016-12-10	2016-12-26	0
BREAD	SUPER_YEAST	2016-12-10	2016-12-26	1
BREAD	SUPER_YEAST	2016-12-24	2016-12-24	0
BREAD	HYPER_YEAST	2016-12-24	2016-12-24	1
CLUB_SANDWICH	MAYO			1
MAYO	EGG			1
MAYO	EGG	2016-12-06	2017-01-01	0
MAYO	EGGS_ORGANIC	2016-12-06	2017-01-01	1
MAYO	OIL			1
MAYO	DIJON_MUSTARD		2016-11-20	1
MAYO	FRENCHS_MUSTARD	2016-11-21		1

MUSTARD	GARLIC_POWDER			1
GARLIC_POWDER	GARLIC			1
CLUB_SANDWICH	BACON			1
CLUB_SANDWICH	BACON	2016-12-31	2016-12-31	0
CLUB_SANDWICH	FAKE_BACON	2016-12-31	2016-12-31	1
FAKE_BACON	HAM			1
FAKE_BACON	BACON_SPICE			1

Amend the API defined in Part 1 to support date-specific queries for components as follows:

`List<Map> getComponents(String productCode, LocalDate date)` for a given product code and date, anywhere in the tree, it returns every component (children) for all levels including or excluding ingredients that are active and inactive for the given date. As before, the returned value should be an ordered list of maps.

```
getComponents("BREAD", "2000-01-01") => [
    { code: "FLOUR", depth: 1, multiplier: 1 },
    { code: "YEAST", depth: 1, multiplier: 1 }
]

getComponents("BREAD", "2016-12-24") => [
    { code: "FLOUR", depth: 1, multiplier: 1 },
    { code: "YEAST", depth: 1, multiplier: 0 },
    { code: "SUPER_YEAST", depth: 1, multiplier: 0 },
    { code: "HYPER_YEAST", depth: 1, multiplier: 1 }
]

getComponents("BREAD", "2016-12-25") => [
    { code: "FLOUR", depth: 1, multiplier: 1 },
    { code: "YEAST", depth: 1, multiplier: 0 },
    { code: "SUPER_YEAST", depth: 1, multiplier: 1 }
]
```

Part3:

As part of automating their ordering process, our client wants to order the basic ingredients based on the forecasted sales of their lunches. Forecasts are supplied as a separate CSV, with a separate entry for each sandwich type and date. As usual, you don't need to implement file I/O. The data is provided by `DataProvider`.

code	**component_code**	**start_date**	**end_date**	**multiplier**
CLUB_SANDWICH	BREAD			1
BREAD	FLOUR			2
BREAD	YEAST			1
BREAD	YEAST	2016-12-10	2016-12-26	0
BREAD	SUPER_YEAST	2016-12-10	2016-12-26	1
BREAD	SUPER_YEAST	2016-12-24	2016-12-24	0
BREAD	HYPER_YEAST	2016-12-24	2016-12-24	0.1
CLUB_SANDWICH	MAYO			3
MAYO	EGG			2
EGG	EGG_WHITE			1
EGG	EGG_YOLK			1
MAYO	OIL			2.5
MAYO	MUSTARD			1
MUSTARD	GARLIC_POWDER			0
MUSTARD	MUSTARD_SEEDS			5
GARLIC_POWDER	GARLIC			1
CLUB_SANDWICH	BACON			4
FRIED_EGGS	EGG			1
FRIED_EGGS	OIL			1

code	**forecast**	**date**
CLUB_SANDWICH	2.0	2017-01-01
FRIED_EGGS	5.0	2017-01-01
CLUB_SANDWICH	4.0	2016-12-24

Building on top of previously implemented code, implement `getForecastsFor()` method for calculating the need for specific basic ingredients for the given date. The output is the orders sent to suppliers for restocking the inventory for the given date. In reality there can be millions of forecasts for different products in different dates so performance should be considered as well.

`List<Map> getForecastsFor(LocalDate date)` for the given date calculate the demand of basic ingredients (i.e. tree leaves) for all active products. Return value should be a list of maps containing the ingredient code (String), amount to order (double) and date (LocalDate).

(Note that the property `amount` *should be a decimal number instead of formula like below. In the example the value has been explicitly shown in as its components for making it easier to understand how it is calculated)*

```
getForecastsFor("2017-01-01") => [
    { code: "FLOUR", amount: 2.0 * 2, date: "2017-01-01" },
    { code: "YEAST", amount: 2.0 * 1.0, date: "2017-01-01" },
    { code: "EGG_WHITE", amount: (2.0 * 3 * 2 * 1) + (5.0 * 1), da
    { code: "EGG_YOLK", amount: (2.0 * 3 * 2 * 1) + (5.0 * 1), dat
    { code: "OIL", amount: (2.0 * 3 * 2.5) + (5.0 * 1), date: "201
    { code: "MUSTARD_SEEDS", amount: 2.0 * 3 * 1 * 5, date: "2017-
    { code: "BACON", amount: 2.0 * 4, date: "2017-01-01" }
]

getForecastsFor("2016-12-24") => [
    { code: "FLOUR", amount: 2.0 * 2, date: "2016-12-24" },
    { code: "HYPER_YEAST", amount: 4.0 * 0.1, date: "2016-12-24" }
    { code: "EGG_WHITE", amount: 2.0 * 3 * 2 * 1, date: "2016-12-2
    { code: "EGG_YOLK", amount: 2.0 * 3 * 2 * 1, date: "2016-12-24
    { code: "OIL", amount: 4.0 * 3 * 2.5, date: "2016-12-24" },
    { code: "MUSTARD_SEEDS", amount: 4.0 * 3 * 1 * 5, date: "2016-
    { code: "BACON", amount: 4.0 * 4, date: "2016-12-24" }
]
```