**Ключевая идея**

Главное отличие версии 2 задачи заключается в использовании функционала класса Optional вместо обрабатываемых (checked) исключений.

Цикл чтения файла и создания мэпов и листа в версии 1 схематически имеет вид:

try(Scanner sc = new Scanner(new FileReader(args[0]));) {

while(sc.hasNext()) {

try {

String line = sc.nextLine();

String [] parts = line.split(Constants.DELIMETR, 2);

if (parts.length < 2){

throw new CsvLineException(Constants.ERROR\_WRONG\_NUMBER, line);

}

WrapperDate date = new WrapperDate(parts[0]);

Purchase purchase = PurchaseFactory.getPurchaseFromFactory(parts[0]);

… //обновление мэпов и листа

} catch (CsvLineException e) {

LOGGER.error("Wrong line: " + e);

}

}

...

} catch (FileNotFoundException e) {

LOGGER.fatal("couldn't open the file: " + args[0] + "\n " + e);

}

Для преобразования цикла в стримовую версию:

1) создайте вспомогательный композитный класс WrapperEntry, содержащий две ссылки на экземпляры WrapperDate и Purchase,

2) используйте следующую идиому для стрима из строк файла:

csvLine => map => Optional<WrapperEntry> => filter => Optional.isPresent() =>

map => Optional.get() => forEach => entry

словами: замапить строку csvLine в экземпляр Optional<WrapperEntry>, фильтровать по непустым экземплярам,

замапить в экземпляр WrapperEntry, для каждого экземпляра entry обновить мапы и лист

try (Stream<String> csvLines = Files.lines(Paths.get(args[0]))) {

//проверьте получившийся java код, выделив 5 строк в комментах ниже:

//---------------- begin

csvLines

.map(WrapperEntry::from)

.filter(Optional::isPresent)

.map(Optional::get)

.forEach(entry -> {

//---------------- end

извлечь геттерами из entry экземпляры WrapperDate и Purchase;

… //обновление мэпов и листа

}

);

… //пункты 5-15 с использованием стримов

} catch (FileNotFoundException e) {

LOGGER.fatal("couldn't open the file: " + args[0] + "\n" + e);

}

Таким образом, для стрима важно, чтобы строки, из которых нельзя выделить валидные сущности, мапились на пустые экземпляры Optional.

**Класс WrapperEntry**

Схематически:

public class WrapperEntry {

private final WrapperDate date;

private final Purchase purchase;

public WrapperEntry(WrapperDate date, Purchase purchase) {

this.date = date;

this.purchase = purchase;

}

...

public static Optional<WrapperEntry> from(String csv) {

String[] values = csv.split(";", 2);

if (values.length < 2) {

log.error(...);

return Optional.empty();

}

Optional<WrapperDate> date = WrapperDate.from(values[0]);

Optional<Purchase> purchase = PurchaseFactory.getPurchase(values[1]);

проверить date и purchase;

возвратить Optional.empty() или экземпляр Optional<WrapperEntry>;

}

}

Таким образом, и в этот класс в альтернативных сценариях, когда scv данные не мапятся в сущности, доставляются не исключения, а пустые сущности, представленные экземплярами Optional.empty().

**Фабричный класс PurchaseFactory**

В предыдущей задаче использовалась реализация фабричного метода с помощью оператора switch. Альтернативный вариант основан на функционале перечислений.

Вспомним, что каждая константа перечисления - это константный вложенный класс. Объявите в перечислении абстрактный метод (достаточно пакетный уровень доступа), возвращающий экземпляр покупки из строки формата csv, а в каждой константе реализуйте этот метод. Тогда фабричный метод станет очень компактным.

public class PurchaseFactory {

private enum PurchaseKind {

@Override

PURCHASE {

Purchase getPurchase(String csv) {

return new Purchase(csv);

}

},

PRICE\_DISCOUNT\_PURCHASE {

...

},

PERCENT\_DISCOUNT\_PURCHASE {

...

};

abstract Purchase getPurchase(String csv);

}

public static Optional<Purchase> getPurchase(String csv) {

try {

String[] values = csv.split(";", 2);

if (values.length < 2) {

log.error(...);

return Optional.empty();

}

преобразовать values[0] в строку codeConvName;

return Optional.of(...(codeConvName).getPurchase(values[1]));

} catch (IllegalArgumentException e) {

log.error(...);

return Optional.empty();

}

}

}

**Классы сущностей**

В этих классах также можно применить функционал Optional. Приватными статическими методами, в которых валидируются аргументы конструктора, возвращайте экземпляры Optional. Например, в классе Byn:

private static **Optional<Integer>** getValidValue(int rubs, int coins) {

if (rubs < 0 || coins < 0 || coins > 99) {

return Optional.empty();

}

return Optional.of(rubs \* 100 + coins);

}

Используя функционал Optional конструктор можно реализовать по схеме:

public Byn(int rubs, int coins) {

this(getValidValue(rubs, coins)

.**orElseThrow**(() -> new WrongArgumentsException(...))

);

}

Примените данный подход в классах всех сущностей.

**Классы необрабатываемых пользовательских исключений**

Можно взять из задачи start1. Альтернативный вариант - это использовать единственный класс WrongArgumentsException, унаследовав его от IllegalArgumentException.

**Класс Runner**

обновить firstPurchasesMap; //2

Используйте putIfAbsent().

обновить dayPurchasesMap; //3

Используйте computeIfAbsent().

вывести каждый мэп с хедером (поясняющим комментарием); //5, 15

Реализуйте printMap() через map.forEach().

вывести в for-each цикле для каждого entry из dayPurchasesMap

значения entry.getKey() и getTotalCost(entry.getValue()); //9

вывести getTotalCost(discountPurchases); //10

Реализуйте getTotalCost(), используя методы стрима map(), reduce() и другие.

Цикл for-each замените на dayPurchasesMap.forEach().

удалить entries; //11-14

Используйте removeIf().