**Класс Byn**

Это класс финансовой величины, который я назвал Byn. Причем по условию класс должен быть [**иммутабельным**](http://www.javapractices.com/topic/TopicAction.do?Id=29).

В классе достаточно определить единственное поле value - значение финансовой величины, выраженное в копейках.

Чем больше конструкторов в классе, тем больше гибкости для создания экземпляров класса. Есть понятие **первичного конструктора**, который должен быть один и множество вторичных. Вторичные конструкторы следует реализовывать через вызов первичного посредством конструкции this().

В простейшем случае в первичном конструкторе каждое поле инициализируется значением соответствующего аргумента:

public Byn(int value) {

this.value = value;

}

Однако, если не все значения аргумента являются допустимыми, то одна из идиом рекомендует выполнить проверку аргумента. В случае недопустимого значения выбрасывается, как правило, экземпляр IllegalArgumentException или его подкласса.

public Byn(int value) {

if (value < 0) {

throw new IllegalArgumentException("...");

}

this.value = value;

}

Создайте следующие вторичные конструкторы:

public Byn() //1

public Byn(Byn byn) //2

public Byn(int rubs, int coins) //3

public Byn(String strByn) //4

Конструктор с вещественным аргументом отсутствует, т.к. ненадежный из-за возможных различных трактовок округления и не подходит для точных расчетов (см. ссылки в документе start1).

Вторичные конструкторы 1 и 2 реализуются тривиально через вызов первичного конструктора. Конструкторы 3 и 4 требуют предварительной проверки аргументов, их преобразования к целому значению, а затем только можно вызвать первичный конструктор. К сожалению, this(...) должен быть первым оператором в конструкторе.

Чтобы исправить данный недостаток, воспользуйтесь следующей идиомой. Проверку аргументов и их преобразование к целому значению оберните в приватный статический метод, возвращающий целое значение. В this(...) передайте вызов этого метода с аргументами конструктора.

public Byn(int rubs, int coins) {

this(getValidValue(rubs, coins));

}

Примените эту идиому для конструкторов 3 и 4. Формат strByn: d+.dd.

Минимально необходимый набор методов для обработки финансовых величин: следующий.

Byn add(Byn byn)

Byn sub(Byn byn)

Byn mul(int k)

Набор переопределяемых методов:

String toString()

boolean equals(Object obj)

public int hashCode()

int compareTo(Byn byn)

Геттер и сеттер для класса финансовой величины лишены смысла. Если их допустить, то они нарушат парадигму инкапсуляции. Сеттер еще невозможен в силу иммутабельности класса.

**Класс Product**

Содержит два **приватных финальных** поля.

public class Product {

private final String name;

private final **Byn** price;

}

Создайте первичный конструктор и вторичные конструкторы по той же схеме, как в классе Byn.

Геттеры для этого класса и классов покупок оправданы.

* Подумайте, какой уровень доступа наиболее приемлемый для них.
* Помните об условии иммутабельности классов, обращая внимание на тип каждого поля (примитивное или ссылочное).

Набор переопределяемых методов:

String toString()

boolean equals(Object obj)

public int hashCode()

Так как оба поля класса являются ссылочными, то надо явно вызывать эти методы по ссылке (name.hashCode(), price.hashCode(), name.equals(other.name), price.equals(other.price)).

Objects.hash(...), Objects.equals() введены в java 8 и по условию - под запретом.

**Класс Purchase**

Содержит два **приватных финальных** поля. Так как использование данного класса не ограничено пакетом, т.е. его могут использовать другие программисты, то применение к данным полям уровня доступа protected не оправдано.

public class Purchase {

private final Product product;

private final int unitsNumber;

}

Для расчета стоимости используйте API класса Byn.

По условию два объекта класса или его подклассов равны, если у них совпадает товар. Следовательно нужно сгенерировать с помощью IDE начальный код методов equals(Object obj) и hashCode() на основе только одного поля product.

Полученный код для большинства тестов будет давать правильный результат.

Однако в нем есть условие, которое устанавливает равенство объектов только из одного и того же класса, что не соответствует условию.

if (**getClass() != obj.getClass()**) {  
 return false;  
 }  
 Выделенный код надо заменить на условие непринадлежности к иерархии класса покупки, используя оператор instanceof.

Детали переопределения метода toString() рассмотрим ниже.

**Класс PriceDiscountPurchase**

В данном подклассе надо объявить одно приватное поле (неконстантное поле экземпляра класса) - ценовую скидку, которая является финансовой величиной.

В переопределенном методе getCost() доступ к полям суперкласса обеспечивается через геттеры. Обратите внимание, что для вызова геттеров ключевое слово super не требуется. Более того, его наличие может привести к проблемам, если потребуется в подклассе переопределить геттер (пусть маловероятно, но все же).

**Метод toString()**

Простая и эффективная схема реализации метода toString() состоит в том, что сначала в суперклассе выводится в csv-строку вся информация об экземпляре суперкласса, а затем в подклассах к ней добавляется информация об экземплярах подклассов.

//в суперклассе

public String toString() {

return product + ";" + unitsNumber + ";" + getCost();

}

//в подклассе

public String toString() {

return super.toString() + ";" + discount;

}

Обратите внимание, что в подклассе при вызове super.toString() будет использоваться версия метода getCost() из соответствующего подкласса, а не суперкласса! Это отличная демонстрация принципа полиморфизма и механизма позднего связывания.

Однако в данной схеме есть недостаток в плане дизайна. Строка экземпляра подкласса сначала содержит исходные данные, затем расчетное значение, далее снова исходное данное. Избавиться от недостатка можно через реализацию в каждом классе protected-метода, который выводит в csv-строку поля экземпляра класса (т.е. только исходные данные без расчетного значения).

//в суперклассе

protected String fieldsToString() {

return product + ";" + unitsNumber;

}

//в подклассе

protected String fieldsToString() {

return super.fieldsToString() + ";" + discount;

}

Тогда реализация метода toString() останется только в суперклассе - нужно сцепить исходные данные и расчетное значение стоимости через символ “;”.

**Фабричный класс PurchasesFactory**

Так как экземпляры классов покупок создаются на основе каких-то внешних признаков, то рекомендуется создать фабричный класс.

Внешний признак в данной задаче - это количество полей в csv-строке:

* если 3 корректных поля, то это экземпляр суперкласса,
* если 4, то подкласса,
* иначе ошибка.

Шаблон Factory приведен в И. Блинов, стр. 606–611, а также стр. 110–112, но хуже. Имеем схему фабричного класса:

public class PurchasesFactory {

public static Purchase getPurchaseFromFactory(**String csv**) {

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

switch(values.length) {

case 3:

return new Purchase(values);

case 4:

return new PriceDiscountPurchase(values);

default :

throw new CsvLineException(...);

}

}

}

Так как конструкторы покупок могут выбрасывать исключения, то оператор switch надо обернуть в try-catch блок и объявить, что фабричный метод выбрасывает проверяемое исключение.

**Классы пользовательских исключений**

Чтобы сохранить информацию о поле класса, которое неправильно инициализируется, имеет смысл создать пользовательские классы исключений, например, NegativeArgumentException, NonpositiveArgumentException, которые будут наследовать класс IllegalArgumentException.

Эти классы должны содержать информацию о причине сбоя, которая пользователю по условию предоставляется в текстовом виде. Но хранить причину в строковом виде – плохое решение. Лучше использовать неправильные исходные целочисленные значения, имена полей и т.п. Часто хорошим решением служит использование перечислений. См. Дж. Блох, статья 45.

В данной задаче могут быть следующие варианты конструкторов:

– public NegativeArgumentException(String cause) – самый простой и одновременно плохой вариант.

– public NegativeArgumentException(int nonpositiveValue, String fieldName) – вариант без контроля поля на этапе трансляции.

– public NegativeArgumentException(int nonpositiveValue, Field field) – вариант с раздельным хранением недопустимого значения и контролем поля.

– public NegativeArgumentException(Field field) – вариант с контролем поля, которое хранит недопустимое значение.

Аналогично для другого подкласса.

Так как во входном файле с большой вероятностью есть ошибочные строки, после которых восстанавливается работа программы, то имеет смысл создать класс пользовательского обрабатываемого исключения CsvLineException. См. Дж. Блох, статьи 40, 43. Полем данного класса целесообразно сделать исключение, связанное с некорректными данными.

**Класс Runner**

Метод main (String[] args), часто называемый раннером, статический. То есть это по своей сути скрипт в процедурном стиле. В нем имеет смысл выносить в методы либо повторяющиеся операторы, либо 5+ операторов, которые решают подзадачу.

Этот класс можно не пакетировать и размещать на верхнем уровне в папке src.

Схема раннера

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

объявить 3 мэпа (lastPurchasesMap, firstPurchasesMap, dayPurchasesMap)

и 1 лист (priceDiscountPurchases);

while (sc.hasNext()) {

String line = sc.nextLine();

рассплитить line на 2 подстроки;

try {

обновить lastPurchasesMap; //1

обновить firstPurchasesMap; //2

обновить dayPurchasesMap; //3

обновить лист discountPurchases; //4

} catch (CsvLineException | WrongDayException e) {  
 залогировать ошибку исходных данных;  
 }

}

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

выполнить поиск покупки в firstPurchasesMap и вывести результат поиска; //6

выполнить поиск покупки в lastPurchasesMap и вывести результат поиска; //7

выполнить поиск MONDAY в dayPurchasesMap и вывести результат поиска; //8

вывести в for-each цикле для каждого entry из dayPurchasesMap

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

вывести getTotalCost(discountPurchases); //10

удалить entries из lastPurchasesMap, где название покупки равно meat; //11

удалить entries из firstPurchasesMap, где день недели равен FRIDAY; //12

удалить entries из firstPurchasesMap,   
 где стоимость покупки меньше 10 руб. и год покупки 2018; //13

удалить entries из dayPurchasesMap, где есть покупка milk; //14

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

} catch (FileNotFoundException e) {  
 вывести сообщение на консоль;  
 залогировать ошибку;  
}

объявление мэпов

В качестве типа данных указывать исключительно интерфейс Map, который должен быть правильно параметризован.

Для покупки необходимо использовать тип Purchase, для даты - любой класс для хранения даты (например, java.util.Date), для дня недели - перечисление (String - это ошибка).

В качестве имплементации мэпов firstPurchasesMap и lastPurchasesMap надо использовать класс HashMap, а для dayPurchasesMap - EnumMap.

Map<Purchase, Date> firstPurchasesMap = new HashMap<>();

Map<Purchase, Date> lastPurchasesMap = new HashMap<>();

Map<Weekday, List<Purchase>> dayPurchasesMap = new EnumMap<>(...);

Так как в качестве ключа первых двух мэпов используется пользовательский класс, то в классе Purchase необходимо согласованно переопределить методы equals() и hashCode().

Причем в классе Purchase в этих методах используется поле ссылочного типа (Product product). Следовательно эти методы также должны быть согласованно переопределены в классе Product и т.д.

Почему эти методы не надо переопределять в перечислении Weekday для третьего мэпа?

---------- ответ в строках ниже ------------

Для третьего мэпа методы переопределять не надо, т.к. в классе Enum, от которого наследуется Weekday, есть правильная реализация этих методов.

-----------------------------------------------------

Обратите внимание, что по условию лист discountPurchases должен быть параметризован подклассом **PriceDiscountPurchase**, а не суперклассом Purchase.

List<**PriceDiscountPurchase**> discountPurchases = new ArrayList<>();

Наконец, чтобы парсить даты и выводить их в том же формате, как они заданы во входном файле, создайте [оберточный класс](https://stackoverflow.com/questions/889160/what-is-a-wrapper-class) WrapperDate, полем которого является экземпляр java.util.Date или аналогичного класса.

Тогда будет

Map<Purchase, **WrapperDate**> firstPurchasesMap = new HashMap<>();

Map<Purchase, **WrapperDate**> lastPurchasesMap = new HashMap<>();

создание экземпляра покупки из подстроки

Используйте фабричный класс, чтобы вынести детали создания экземпляра за пределы раннера.

обновить lastPurchasesMap; //1

обновить firstPurchasesMap; //2

Изучите функционал метода put() интерфейса Map. Для реализации пунктов 1 и 2 достаточно 4 строки кода, причем строка 4 - это конец блока (т.е. }).

обновить dayPurchasesMap; //3

Необходимо минимизировать количество операций с мэпом.

В тьюториале Дж. Блох приводит очень красивую идиому создания мультимэпа.

См. <https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html>, пункт Multimaps.

Для обращения к мэпу используются всего лишь два вызова: один вызов get() и один put().

обновить лист discountPurchases; //4

Выполняется после тривиальной проверки принадлежности экземпляра класса к подклассу.

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

Очевидно, что нужно создать приватный статический метод.

Внутри этого метода должны быть три вызова вызова еще одного и того же метода printMap(**Map** map, String message).

, где Map - raw type (антипаттерн!!!).

Этот метод необходимо параметризовать, чтобы избавиться от антипаттерна.

Получаем следующий заголовок

private static <K, V> void printMap(Map<K, V> map, String message)

Обратите внимание, что в мэпах в качестве типов-параметров используются буквы K и V.

Для вывода пар ключ-значение используйте идиому из тьюториала, где в for-each цикле итерация идет по entry.

Альтернативный вариант объявления метода

private static void printMap(Map<**?, ?**> map, String message)

уступает в типобезопасности основному варианту.

выполнить поиск покупки в firstPurchasesMap и вывести результат поиска; //6

выполнить поиск покупки в lastPurchasesMap и вывести результат поиска; //7

выполнить поиск MONDAY в dayPurchasesMap и вывести результат поиска; //8

Очевидно, что и эти три действия должны быть реализованы путем вызова одного и того же метода.

Создайте параметризованный метод **поиска** по ключу-параметру **и вывода** найденного значения.

---------- заголовок метода -----------------

private static <**K**, V> void findAndShow(Map<**K**, V> map, **K** searchKey, String header)

-----------------------------------------------------

Учтите, что значение может отсутствовать и должно быть понятное сообщение.

Данный метод не требует цикла, надо использовать функционал мэпа, т.е. методы, объявленные в интерфейсе Map.

Всегда нужно стремиться минимизировать количество обращений к коллекции. Учитывая, что в мэпах этой задачи нет null-значений, то вызов метода get() можно использовать как для проверки существования ключа, так и получения значения.

вывести в for-each цикле для каждого entry из dayPurchasesMap

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

вывести getTotalCost(discountPurchases); //10

Неправильный заголовок метода:

private static Byn getTotalListCost(List<**Purchase**> purchases)

Чтобы не было ошибки компилятора, где-то придется использовать даункаст, что есть злостный антипаттерн.

Следовательно метод надо параметризовать и ограничить тип.

private static <T extends Purchase> Byn getTotalCost(List<T> purchases)

Избавьтесь от типа в заголовке метода

private static ~~<T extends Purchase>~~ Byn getTotalCost(List<T> purchases)

---- подсказка в следующей строке ----

примените ? в типе аргумента

-----------------------------------------------------

удалить entries из lastPurchasesMap, где название покупки равно meat; //11

удалить entries из firstPurchasesMap, где день недели равен FRIDAY; //12

Сначала рассмотрим пункты 11 и 12. Неправильные заголовки метода удаления:

private static void remove(Map<Purchase, WrapperDate> map, **String** pattern)

private static void remove(Map<Purchase, WrapperDate> map, **Object** pattern)

private static **<T>** void remove(Map<Purchase, WrapperDate> map, **T** pattern)

Конечно, можно задать необходимый экземпляр для сравнения в строке, в подклассе Object, в экземпляре параметризованного типа.

Но плохо, что алгоритм метода удаления зависит от задания, т.е. заточен под конкретное условие.

А если надо удалить по стоимости?

А если еще по стоимости и году одновременно (пункт 13)?

удалить entries из firstPurchasesMap,   
 где стоимость покупки меньше 10 руб. и год покупки 2018; //13

В универсальной реализации метода удаления детали проверки покупки на удаление нужно вынести за пределы метода с помощью имплементации интерфейса

public interface EntryChecker {

boolean check(Map.Entry<**Purchase, WrapperDate**> entry);

}

Тогда метод удаления имеет заголовок

private static void removeEntries(Map<Purchase, WrapperDate> map, **EntryChecker checker**)

В теле цикла, если метод check() возвращает true, надо удалить entry.

В идиоме удаления элементов мэпа явно используется итератор.

См. <https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html>, пункт Collection Views.

Не создавайте классы чекеров.

Имплементации интерфейса EntryChecker сделайте анонимными классами.

Также не создавайте именованные экземпляры анонимных чекеров, т.к. они висят в памяти до конца метода.

удалить entries из dayPurchasesMap, где есть покупка milk; //14

Остался последний шаг, чтобы обобщить удаление. Очевидно, надо параметризовать метод удаления.

---------------- подсказка ниже -------------

//параметризуйте интерфейс и метод удаления в раннере

public interface EntryChecker <K, V> {

boolean check(Map.Entry<K, V> entry);

}

private static <K, V> void removeEntries(Map<K, V> map, EntryChecker<K, V> checker) {

…

}

-----------------------------------------------------