# Training materials

* The Java Tutorials. [Lesson: Generics](https://docs.oracle.com/javase/tutorial/java/generics/index.html).
* The Java Tutorials. [Lesson: Exceptions](https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html).
* The Java Tutorials. [Trail: Collections](https://docs.oracle.com/javase/tutorial/collections/index.html).

# Code Exercise

Create the package named by.epam.inner for immutable classes of entities.

1. Byn - a financial value for BYN (belarusian rubles).

String representation of a financial value has the format:

d+.dd

1. Product with two fields: name and price in BYN.

String representation of a product has the csv format:

name;price

1. Superclass Purchase that represents a product purchase. It has two fields: product and number of purchased units.

String representation of a purchase has the csv format:

product;number;cost

where cost is a purchase cost in Byn, and cost = price \* number.

1. Subclass PriceDiscountPurchase that represents a product purchase with a price discount. It has one field: price discount.

String representation of a purchase has the csv format:

product;number;discount;cost

where cost = (price - discount) \* number.

Any two purchases are equal if their products are equal.

Any two products are equal if their names and prices are equal.

A csv file contains a series of text lines. Every line contains information about a purchase date and a purchase itself in the format:

date;name;price;number

or

date;name;price;number;discount

A date format: yy-mm-dd, where yy - two last digits of a year of this century, mm is for month and dd is for day.

A name format: any non empty string.

A price and a discount format: d+.dd, but not 0.00.

A number format: d+, but not 0.

Wrong lines in a csv file are possible. Log wrong lines during the process of loading a content of a csv file with the help of Log4j api. A record must contain a wrong line itself and the reason(s) of error.

Define the Runner class in the default package. An argument of the Runner is the name of a csv file.

The algorithm of the method main( ):

1. load the content of a csv file into **the map** where a purchase is a key and a date of a last purchase in a csv file is a value;
2. load the content of a csv file into **the map** where a purchase is a key and a date of a first purchase in a csv file is a value;
3. load the content of a csv file into **the enumerated map** where a weekday is a key and purchases list for this weekday is a value;
4. load instances of the subclass PriceDiscountPurchase from a csv file into **the** **List**<**PriceDiscountPurchase**> **discountPurchases**;
5. output maps to the console;
6. find and output the dates for bread with price 1.55 in the second map;
7. find and output the date for bread with price 1.70 in the first map;
8. find and output all purchases on MONDAY;
9. output the total cost of all purchases for each weekday;
10. output the total cost of purchases from the list **discountPurchases**;
11. remove all entries from the first map where the purchase name is meat;
12. remove all entries from the second map on FRIDAY;
13. remove all entries from the second map where the cost is less than   
    10.00 BYN and year is 2018;
14. remove all entries from the enumerated map where a product name is milk in some purchase;
15. output maps to the console.

The example of a csv file:

19-02-15;bread;1.55;1;0.20

20-01-03;milk;1.31;2

19-02-15;bread;1.50

20-03-24;bread;1.54;3

18-11-30;bread;1.55;5

11-30;bread;1.55;5

potato;0.80;2;0.10

20-02-10;potato;0.80;2;0.10

90-02-15;bread;1.50;1

20-03-24;butter;2.70;1

20-03-24;butter;2.41;1;0.50

19-02-15;bread;1.50;1;1

19-20-15;bread;2.50;2

19-20-15;bread;2.50;-2

20-01-03;meat;8.00;2;0.90

18-11-30;potato;0.80;5;0.10

19-12-15;milk;1.31;2

20-01-03;bread;1.54;3

20-05-10;bread;1.55;5

20-04-12;meat;9.00;2;0.70

20-01-03;potato;0.80;3;0.20

Требования к проекту

– Использовать фреймворк maven.

– Покрыть код junit тестами.

Ограничения и замечания к задаче 1

– Данная задача предназначена для повторения параметризованных типов, исключений и коллекций.

– **Запрещен** функционал **java 8** (стримы, лямбды, появившиеся классы и интерфейсы).

– **Запрещены** [raw types](https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html). Как следствие, не использовать класс Object. Это задача как по теме [Collections framework](https://docs.oracle.com/javase/tutorial/collections/index.html), так и [**Generic Types and Methods**](https://docs.oracle.com/javase/tutorial/java/generics/index.html) **(!!!)**.

– В данной задаче на все финансовые величины ввожу следующее ограничение: **максимальное значение стоимости в копейках не больше 0x7fffffff** (т.е. максимум диапазона int).

– Запрещаю использовать функционал класса BigDecimal.

– Читайте у Джошуа Блоха раздел "Если требуются точные ответы, избегайте использования типов float и double". Следствие: **вещественный (плавающий) тип не подходит для финансовых расчетов**. Вот еще один убедительный аргумент - ссылка на статью [Representing money](http://www.javapractices.com/topic/TopicAction.do?Id=13) от авторов отличного ресурса [Java practices](http://www.javapractices.com/home/HomeAction.do).

– Создать три мэпа (пункты 1, 2, 3) и лист (пункт 4) за **один** проход по файлу (в одном цикле и без дополнительных структур данных).

– Критерий дня первой/последней (first/last) покупки зависит от очередности появления **в файле**, а не по правилу первая покупка - это та покупка, которая самая ранняя, а последняя - та, которая самая поздняя.

– Создать **единый** приватный статический метод раннера для вывода мэпов на консоль. Формат вывода - последовательность строк вида key => value, см. примеры ниже.

– Создать **единый** приватный статический метод раннера для поиска элемента в мэпе (пункты 6, 7, 8).

– Для вычисления суммы покупок по листу создать **единый** статический метод раннера и использовать его для пунктов 9 и 10.

– Создать **единый** приватный статический метод раннера для удаления элементов из мэпа (пункты 11 - 14).

– Не забыть проверить решение, когда нет исходного файла.

– После чтения вышеприведенного файла должны быть получены следующие мэпы:

**map where date of a first purchase is a value**

butter;2.41;1;0.50;1.91 => 20-03-24

bread;1.54;3;4.62 => 20-03-24

bread;1.55;1;0.20;1.35 => 19-02-15

meat;9.00;2;0.70;16.60 => 20-04-12

milk;1.31;2;2.62 => 20-01-03

meat;8.00;2;0.90;14.20 => 20-01-03

butter;2.70;1;2.70 => 20-03-24

potato;0.80;2;0.10;1.40 => 20-02-10

bread;1.50;1;1.50 => 90-02-15

**map where date of a last purchase is a value**

butter;2.41;1;0.50;1.91 => 20-03-24

bread;1.54;3;4.62 => 20-01-03

bread;1.55;1;0.20;1.35 => 20-05-10

meat;9.00;2;0.70;16.60 => 20-04-12

milk;1.31;2;2.62 => 19-12-15

meat;8.00;2;0.90;14.20 => 20-01-03

butter;2.70;1;2.70 => 20-03-24

potato;0.80;2;0.10;1.40 => 20-01-03

bread;1.50;1;1.50 => 90-02-15

**enumerated map**

MONDAY => [potato;0.80;2;0.10;1.40]

TUESDAY => [bread;1.54;3;4.62, butter;2.70;1;2.70, butter;2.41;1;0.50;1.91]

WEDNESDAY => [bread;1.50;1;1.50]

FRIDAY => [bread;1.55;1;0.20;1.35, milk;1.31;2;2.62, bread;1.55;5;7.75, meat;8.00;2;0.90;14.20, potato;0.80;5;0.10;3.50, bread;1.54;3;4.62, potato;0.80;3;0.20;1.80]

SUNDAY => [milk;1.31;2;2.62, bread;1.55;5;7.75, meat;9.00;2;0.70;16.60]

# Guidelines

**Класс Byn**

Это класс финансовой величины, который назвем 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. В силу ограничения на диапазон тип поля - int.

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 пользовательского подкласса IllegalArgumentException.

public Byn(int value) {

if (value < 0) {

throw new NegativeArgumentException("...");

}

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.

Условия валидности в конструкторе 3:

* рубли и копейки не меньше 0 и копейки меньше 100.
* rubs \* 100 + coins <= максимальное значение поля value в классе Byn.

Условие валидности в конструкторе 4: формат strByn: d+.dd.

Для конструктора 4 используйте предварительно скомпилированное регулярное выражение. Если аргумент strByn ему не соответствует, то выбросьте экземпляр PatternArgumentException - пользовательского подкласса IllegalArgumentException.

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 невозможен еще в силу иммутабельности класса.

**Классы необрабатываемых 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.

Полная иерархия необрабатываемых исключений для данной задачи:

* NonpositiveArgumentException -> NegativeArgumentException -> IllegalArgumentException
* RangeArgumentException -> IllegalArgumentException
* PatternArgumentException -> IllegalArgumentException
* EmptyArgumentException -> IllegalArgumentException

В зависимости от специфики реализации допустимы другие подклассы IllegalArgumentException.

**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.

Условие валидности: непустое имя товара и ненулевая цена.

Геттеры для этого класса и классов покупок оправданы.

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;

}

Создайте первичный конструктор и вторичные конструкторы, контролируя их аргументы.

Условия валидности:

* положительное количество товара,
* цена \* количество <= максимальное значение поля value в классе Byn (не факт, что это максимум диапазона int).

В конструкторе из массива строк проверка длины массива обязательная, т.к. запрос на создание экземпляра покупки может прийти из любого класса, а не только из фабричного класса, где проверяется длина массива. Аналогично в конструкторе из csv строки надо проверить количество аргументов после расщепления csv строки, а далее попытаться создать экземпляр покупки из строковых аргументов.

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 и которая должна быть: а) ненулевой, б) меньше цены - это условия валидности.

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**) **throws CsvLineException** {

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

try {

switch(values.length) {

case 3:

return new Purchase(values);

case 4:

return new PriceDiscountPurchase(values);

default :

throw new CsvLineException(...);

}

} catch (IllegalArgumentException e) {

throw new CsvLineException(...);

}

}

}

Как видно из данного кода, в классах покупок должны быть валидные конструкторы, принимающие массив строк.

Согласно условия во входном файле могут быть ошибочные строки, после которых восстанавливается работа программы. Следовательно, нужно создать класс пользовательского **обрабатываемого** исключения CsvLineException. Полем данного класса целесообразно сделать исключение, связанное с некорректными данными (подкласс IllegalArgumentException).

Таким образом, конструкторы покупок выбрасывают **необрабатываемые** пользовательские исключения, причем все они - экземпляры подклассов класса IllegalArgumentException. Именно в этом методе их все надо перехватить, обернув оператор switch в try-catch блок и объявив, что фабричный метод выбрасывает экземпляр **обрабатываемого** исключения CsvLineException.

**Класс WrapperDate**

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.

**Вторичный** конструктор со строковым аргументом должен выбрасывать экземпляр **необрабатываемого** исключения PatternArgumentException.

**Класс WrapperEntry**

Код в раннере можно сделать очень компактным, создав обертку над парой (дата;покупка). В этом классе надо объявить два ссылочных поля: WrapperDate и Purchase.

Детали создания экземпляров спрятаны в статическом методе валидации, который вызывается из **вторичного** конструктора со строковым аргументом. Этот конструктор должен выбрасывать экземпляр **обрабатываемого** исключения CsvLineException.

public class WrapperEntry {

private final WrapperDate date;

private final Purchase purchase;

...

public static WrapperEntry gerValidEntry(String csv) throws CsvLineException {

String[] values = csv.split(";", 2); //рассплитить line не более, чем на 2 подстроки

if (values.length < 2) {

throw new CsvLineException(...);

}

создать экземпляр WrapperDate date из values[0];

создать экземпляр Purchase purchase из values[1];

return new WrapperEntry(date, purchase);

}

}

**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 лучше 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();

try {

WrapperEntry entry = new WrapperEntry(line);

извлечь экземпляры date и purchase из entry;

обновить lastPurchasesMap; //1

обновить firstPurchasesMap; //2

обновить dayPurchasesMap; //3

обновить лист discountPurchases; //4

} catch (CsvLineException 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 - WrapperDate, для дня недели - перечисление (String - это ошибка).

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

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

Map<Purchase, WrapperDate> 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<>();

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) {

…

}

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

# Процедура принятия

Этапы выполнения задачи 1

Этап 0. Знакомство с условием.

~~– Заполните~~ [~~форму регистрации дедлайна~~](https://docs.google.com/forms/d/e/1FAIpQLSe2ph0LJpVgTGXaE9_xLnbucIFF4MORP34HwdG5g1GYRfcYNg/viewform) ~~первой версии.~~

Этап 1. Реализация первой версии решения.