# Training materials

* The Java Tutorials. [Lambda Expressions](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html).
* The Java Tutorials. [Method References](https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html).
* The Java Tutorials. [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html). Subsection Aggregate Operations.
* The Java Tutorials. [Lesson: Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html).
* The Java Tutorials. [Reduction](https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html).
* The Java Tutorials. [The Reflection API](https://docs.oracle.com/javase/tutorial/reflect/index.html).

# Code Exercise

Create the superclass Trial for a trainee trial. Use the only field account to present a trainee. Any trial consists of two tests to be estimated by integer values (mark1 and mark2) from 0 to 100 inclusively. A trial is considered to be passed if the sum of marks is not less than some class constant.

Create subclasses for following kinds of trials:

⎼ LightTrial - simplified (or light) trial. It also includes two tests. But a trial is passed if both marks are not less than some constants for every test.

⎼ StrongTrial - complicated (or strong) trial with two tests too. A trial is passed if the sum of a half mark1 and a whole mark2 is not less than the same constant as in the base trial.

⎼ ExtraTrial - extraordinary (or extra) trial. It contains an additional test. This trial is passed if the base trial is passed and a mark3 for an additional test is not less than some constant.

A totally valid json file contains an array of objects in the format:

[

{

class: value, //"Trial", "LightTrial" or "StrongTrial"

args: {

account: value,

mark1: value,

mark2: value

}

},

…,

{

class: "ExtraTrial”,

args: {

account: value,

mark1: value,

mark2: value,

mark3: value

}

},

...

]

If some object contains extra data, it is considered correct. This data must be registered at the WARN level. Otherwise an object is not correct, and it must be registered at the ERROR level.

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. Create the List implementation from a json file. Use [Gson library](https://github.com/google/gson) to parse it and to identify trial entities.

2. Print the collection content (one element per line).

3. Print the number of passed trials.

4. Sort the collection by the sum of first and second marks.

5. Print sums of first and second marks from the collection (one sum per line).

6. Create a new collection from unpassed trials, clear all marks and print this collection. Check whether all trials are failed (the result type is boolean).

7. Create a string from sums of first and second marks of sorted collection (see item 4) in the format:

sum[0], sum[1], … , sum[sum.length - 1]

#### Замечания и ограничения

– Запрещается валидировать экземпляры классов в конструкторах. Для этой цели используйте фабричный класс (или классы).

– Для обработки коллекции используйте стримы.

# Guidelines

**Класс Runner**

пункт 1.

Реализуйте в 2 этапа.

1. Создание листа экземпляров JsonObject:

Type type = new TypeToken<List<JsonObject>>(){}.getType();

List<JsonObject> jsonObjects = new Gson().fromJson(new FileReader(args[0]), type);

2. Маппинг листа jsonObjects в лист экземпляров Trial по аналогии с задачей start2:

List<Trial> trials = jsonObjects.stream()

.map(...) //см. ниже об аргументе метода

.filter(Optional::isPresent)

.map(Optional::get)

.collect(Collectors.toList());

Реализация аргумента метода map() рассмотрена ниже.

пункт 2.

тривиально через method reference.

пункт 3.

аналогично 2.

пункт 4.

используйте метод sort() у колекции.

аргумент метода - компаратор, инициализированный лямбдой.

пункт 5.

intermediate operation - метод map у стрима, в который передается та же лямбда, как в пункте 4.

копи-паст - антипаттерн.

очевидное решение - реализовать в суперклассе дополнительный **финальный** метод и использовать method reference.

более изящное решение - создать ссылку на лямбду и использовать ее в пунктах 4 и 5 (и далее еще в 7).

пункт 6.

выполнение данного пункта через 3+ оператора не представляет трудностей.

минимально возможное количество и при этом изящно - 2 оператора.

даункасты - антипаттерн, поэтому предварительно надо в суперклассе определить метод:

1. clearMarks() и переопределить его в подклассе ExtraTrial,
2. getCopy() и переопределить его в каждом подклассе.

в первом операторе после извлечения стрима из коллекции надо применить операции filter(), map(), дважды peek() и, наконец, collect() для создания новой коллекции.

второй оператор нужен для подтверждения факта, что все испытания в новой коллекции провалены.

пункт 7.

оператор 1 - преобразование коллекции в целочисленный массив через операцию mapToInt().

оператор 2 - преобразование массива в строковую коллекцию и ее дальнейшее обработка через функционал класса Collectors.

**Реализация аргумента метода map() для пункта 1**

Аргументом метода map() может быть ссылка на статический метод фабричного класса, аргументом которого является экземпляр JsonObject:

public static Optional<Trial> getTrial(JsonObject jsonObject)

В теле метода получите из jsonObject необходимые для создания экземпляров Trial и его подклассов аргументы.

Например,

String className = jsonObject.get("class").getAsString();

JsonObject args = jsonObject.get("args").getAsJsonObject();

Проанализируйте className и аргументы. Если они валидные, то возвратите Optional.of(trial), иначе Optional.empty(), где trial - это экземпляр Trial или его подкласса.

**Классы сущностей**

Это тривиальные POJO классы, которые содержат поля, конструктор общего назначения, геттеры, сеттеры и другие “очевидные” методы.

С помощью конструктора такого класса можно создать невалидный экземпляр. Поэтому их валидация должна быть во внешних сущностях, как правило, фабричных классах.

Ответственность за создание экземпляров POJO классов не напрямую, а через фабричные методы кладется на программиста.

**Классы валидаторов (очевидный способ)**

public class TrialValidator {

private final Trial trial;

...

public TrialValidator(Class<? extends Trial> trialClass) {

инициализировать trial, используя trialClass;

}

public Trial getValidTrial(JsonElement element) { //как вариант, Optional<Trial>

JsonObject jsonObject = element.getAsJsonObject();

проверить количество аргументов;

checkArgs(jsonObject);

setFields(jsonObject); //как вариант, можно объединить с методом checkArgs()

return getTrial();

}

protected Trial getTrial() {

return trial;

}  
}

Создайте подкласс(ы) валидатор-а(-ов), переопределив методы checkArgs(), setFields() (и другие?).

public class ExtraTrialValidator extends TrialValidator {

private ExtraTrial extraTrial;

...

//способ 1

public ExtraTrialValidator(Class<? extends ExtraTrial> trialClass) {

инициализировать extraTrial, используя trialClass;

//недостаток способа - копи-паст конструктора суперкласса.

}

//способ 2

// public ExtraTrialValidator(Class<? extends ExtraTrial> trialClass) {

// super(trialClass);

// }

//недостаток способа - валидация на разных сущностях.

//сначала выполняется на экземпляре суперкласса,

//затем в методе setFields() можно создать экземпляр подкласса:

// extraTrial = new ExtraTrial(super.getTrial(), mark3);

…

protected Trial getTrial() {

return extraTrial;

}  
}

**Фабричный класс**

public class TrialFactory {

...

private static final TrialDeserializer TRIAL\_DESERIALIZER = new TrialDeserializer();

private static final Gson GSON = new GsonBuilder()

.registerTypeAdapter(Trial.class, TRIAL\_DESERIALIZER)

.registerTypeAdapter(LightTrial.class, TRIAL\_DESERIALIZER)

.registerTypeAdapter(StrongTrial.class, TRIAL\_DESERIALIZER)

.registerTypeAdapter(ExtraTrial.class, TRIAL\_DESERIALIZER)

.create();

public static Optional<Trial> getTrial(JsonObject jsonObject) {

try {

...

return Optional.of(GSON.fromJson(args, classType));

} catch (...) {

return Optional.empty();

}

}

}

**Десериализатор**

public class TrialDeserializer implements JsonDeserializer<Trial> {

...

private enum TrialKind {

TRIAL(new TrialValidator(Trial.class)),

...;

private final TrialValidator validator;

TrialKind(TrialValidator validator) {

this.validator = validator;

}

Trial getTrial(JsonElement element) {

return validator.getValidTrial(element);

}

}

@Override

public Trial deserialize(JsonElement element,

Type type,

JsonDeserializationContext context) {

...

return TrialKind.valueOf(trialKind).getTrial(element);

}

}

**Классы валидаторов (способ 2)**

Оба способа, рассмотренных выше, имеют свой недостаток.

Эффективное решение - это создать параметризованный валидатор.

public class TrialValidator<T extends Trial> {

private final T trial;

...

public TrialValidator(<T> trial) {

this.trial = trial;

}

public Trial getValidTrial(JsonElement element) { //как вариант, Optional<Trial>

JsonObject jsonObject = element.getAsJsonObject();

проверить количество аргументов;

checkArgsAndSetFields(jsonObject);

return trial;

}

protected T getTrial() {

return trial;

}  
 ...

}

Подкласс ExtraTrialValidator нужно параметризовать классом ExtraTrial, и тогда код класса существенно сокращается. Проверьте самостоятельно.

Предусмотрите случай расширения иерархии валидаторов. Т.е. как правильно объявить ExtraTrialValidator при условии, что у класса ExtraTrial может быть подкласс?

Создание экземпляров валидаторов реализуется во вложенном в десериализатор перечислении TrialKind следующим образом.

private enum TrialKind {

TRIAL(new TrialValidator<>(new Trial())),

LIGHT\_TRIAL(new TrialValidator<>(new LightTrial())),

...;

...

}