# FilterProcessor

Ниже приведено детальное описание логики работы класса FilterProcessor из примера кода.

**Общая идея**

Класс FilterProcessor отвечает за непосредственно обработку (фильтрацию) данных из входных файлов, а также за запись результатов в выходные файлы и сбор статистики. Он тесно взаимодействует с:

1. **Настройками (Options)** – где хранятся флаги и параметры, переданные через аргументы командной строки.
2. **Коллекторами статистики:**

* NumberStatCollector integerStats для целых чисел.
* NumberStatCollector floatStats для вещественных чисел.
* StringStatCollector stringStats для строк.

1. Потоками записи (по одному на каждый тип данных). Создаются «по требованию», то есть только тогда, когда в процессе парсинга встретился соответствующий тип данных.

Таким образом, FilterProcessor последовательно «прогоняет» все входные файлы, читает их по строкам, определяет тип каждой строки (целое, вещественное или строка), записывает её в соответствующий выходной файл (при необходимости – в режиме добавления), и параллельно обновляет статистику. В конце всей обработки поток записи для каждого типа закрывается, а собранная статистика выводится в консоль.

**Пошаговая логика**

1. **Поля класса**

**private final Options options;**

Хранит все настройки, переданные утилите (**-a**, **-s**, **-f**, **-o**, **-p** и список входных файлов).

**private final NumberStatCollector integerStats = new NumberStatCollector();**

**private final NumberStatCollector floatStats = new NumberStatCollector();**

**private final StringStatCollector stringStats = new StringStatCollector();**

Здесь хранятся объекты для сбора статистики. Для каждого типа свой коллектор (отвечает за подсчёт количества, минимума, максимума и т.д.).

**private boolean hasIntegers = false;**

**private boolean hasFloats = false;**

**private boolean hasStrings = false;**

Флаги, показывающие, встречались ли в процессе обработки данные соответствующего типа. Нужны для того, чтобы:

* не создавать файл для типа данных, который так и не встретился во входных файлах (чтобы не плодить пустые файлы).
* выводить статистику только по тем типам, которые реально появились.

**private BufferedWriter intWriter = null;**

**private BufferedWriter floatWriter = null;**

**private BufferedWriter stringWriter = null;**

Потоки записи для каждого типа. Изначально null; создаются при первом же обнаружении данных соответствующего типа.

1. **Метод process()**

public void process() {

for (String inputFile : options.getInputFiles()) {

processSingleFile(inputFile);

}

closeWriters();

printStats();

}

1. Перебирает список входных файлов, указанных в options.
2. Для каждого файла вызывает **processSingleFile(inputFile)**.
3. После обработки всех файлов вызывает **closeWriters()** (закрыть все открытые потоки записи).
4. И в завершение выводит статистику через **printStats()**.
5. **Метод processSingleFile (String inputFile)**

private void processSingleFile(String inputFile) {

try (BufferedReader br = new BufferedReader(new FileReader(inputFile))) {

String line;

while ((line = br.readLine()) != null) {

line = line.trim();

if (line.isEmpty()) {

continue;

}

classifyAndWrite(line);

}

} catch (FileNotFoundException e) {

System.err.println("Файл не найден: " + inputFile + ". Ошибка: " + e.getMessage());

} catch (IOException e) {

System.err.println("Ошибка ввода-вывода при чтении " + inputFile + ": " + e.getMessage());

}

}

1. Открывает входной файл через **BufferedReader**.
2. Читает файл построчно.
3. Каждую строку предварительно обрезает методом **trim()**, чтобы убрать пробелы/табуляции по краям.
4. Пропускает пустые строки (если **line.isEmpty()**).
5. Для непустых строк вызывает **classifyAndWrite(line)**, где происходит классификация (определение типа) и запись.
6. В случае ошибок чтения или отсутствия файла печатает сообщение об ошибке, но продолжает работу с другими файлами (частичная обработка).

Обратите внимание, что блок **try-with-resources** автоматически закрывает **BufferedReader** после обработки файла или при возникновении исключения.

1. **Метод classifyAndWrite (String line)**

private void classifyAndWrite(String line) {

if (isInteger(line)) {

writeInteger(line);

} else if (isFloat(line)) {

writeFloat(line);

} else {

writeString(line);

}

}

1. Сначала проверяет, является ли строка **целым числом**, вызывая **isInteger(line)**.
2. Если не целое, проверяет, **вещественное** ли это число, методом **isFloat(line)**.
3. Если не прошло оба предыдущих условия – **записываем как строку**.
4. **Методы isInteger(String s) и isFloat(String s)**

private boolean isInteger(String s) {

try {

Integer.parseInt(s);

return true;

} catch (NumberFormatException e) {

return false;

}

}

* Пытается распарсить значение как **Integer**.
* Если получилось – возвращает **true**.
* Если выброшено исключение, значит это **не целое число**.

private boolean isFloat(String s) {

try {

Double.parseDouble(s.replace(",", "."));

return true;

} catch (NumberFormatException e) {

return false;

}

}

* Аналогично, только тут используется **Double.parseDouble** (и поддерживаются экспоненциальные форматы, вроде 1.5E-10).
* Дополнительно заменяем запятую **(',')** на точку **('.')** – если вдруг строка содержит десятичную запятую.

Обратите внимание, что в примере любой нецелый формат распознаётся как «вещественное число» (double).

1. **Методы записи: writeInteger, writeFloat, writeString**

Каждый из них делает однотипные действия:

1. Проверяет, создан ли уже соответствующий поток записи (по флагу **hasIntegers/hasFloats/hasStrings**).
2. Если поток ещё не создан, вызывает **createWriter(getTargetFileName(...))**.
3. Пишет строку и добавляет перевод строки.

Обновляет статистику (коллектор).

Например, для целых:

private void writeInteger(String line) {

if (!hasIntegers) {

hasIntegers = true;

intWriter = createWriter(getTargetFileName("integers.txt"));

}

try {

intWriter.write(line);

intWriter.newLine();

intWriter.flush();

double value = Double.parseDouble(line);

integerStats.addValue(value);

} catch (IOException e) {

System.err.println("Ошибка записи целого числа: " + e.getMessage());

}

}

Аналогично для вещественных и для строк – только записываем в другие файлы и обновляем соответствующие коллекты статистики.

Обратите внимание, что и для целых, и для вещественных в статистике используется **Double.parseDouble**, поэтому в **NumberStatCollector** данные хранятся как **double**.

1. **Метод createWriter(String fileName)**

private BufferedWriter createWriter(String fileName) {

try {

Path path = Paths.get(fileName);

Files.createDirectories(path.getParent());

return new BufferedWriter(new FileWriter(fileName, options.isAppendMode()));

} catch (IOException e) {

System.err.println("Не удалось создать файл для записи: " + fileName + ". " + e.getMessage());

return null;

}

}

1. Формирует Path из имени файла.
2. Создаёт нужные директории, если их не было (**createDirectories**).
3. Открывает **FileWriter** в режиме append или overwrite (зависит от **options.isAppendMode()**).
4. Оборачивает его в **BufferedWriter** и возвращает.
5. Если произошла ошибка, пишет сообщение в консоль и возвращает **Null**.

Таким образом, каждый поток записи создаётся только единожды – при первом встретившемся элементе нужного типа.

1. **Метод getTargetFileName (String defaultName)**

private String getTargetFileName(String defaultName) {

String dir = options.getOutputPath();

if (!dir.isEmpty() && !dir.endsWith(File.separator)) {

dir = dir + File.separator;

}

return dir + options.getPrefix() + defaultName;

}

1. Берёт базовый путь -o, если он задан в **options.getOutputPath()**.
2. При необходимости добавляет к нему системный разделитель каталогов (**File.separator**), если там не стоит «/» или «\».
3. Склеивает префикс **-p** (если есть) и дефолтное имя, например, **integers.txt**.
4. Получается полный путь для выходного файла. Пример:  
   Если **-o /some/path и -p my\_**, то результат для целых: **/some/path/my\_integers.txt**.
5. **Метод closeWriters()**

private void closeWriters() {

try {

if (intWriter != null) intWriter.close();

if (floatWriter != null) floatWriter.close();

if (stringWriter != null) stringWriter.close();

} catch (IOException e) {

System.err.println("Ошибка при закрытии файлов: " + e.getMessage());

}

}

Закрывает потоки записи (если они были открыты). Выполняется в конце, чтобы корректно освободить ресурсы, дописать данные в файлы и т. д.

1. **Метод printStats()**

private void printStats() {

System.out.println("=== Statistics ===");

if (hasIntegers) {

System.out.println("-- Integers --");

if (options.isFullStats()) {

System.out.println(integerStats.getFullStats());

} else if (options.isShortStats()) {

System.out.println(integerStats.getShortStats());

} else {

System.out.println(integerStats.getShortStats());

}

}

if (hasFloats) {

System.out.println("-- Floats --");

if (options.isFullStats()) {

System.out.println(floatStats.getFullStats());

} else if (options.isShortStats()) {

System.out.println(floatStats.getShortStats());

} else {

System.out.println(floatStats.getShortStats());

}

}

if (hasStrings) {

System.out.println("-- Strings --");

if (options.isFullStats()) {

System.out.println(stringStats.getFullStats());

} else if (options.isShortStats()) {

System.out.println(stringStats.getShortStats());

} else {

System.out.println(stringStats.getShortStats());

}

}

}

1. Печатает заголовок **«=== Statistics ===»**.
2. Проверяет каждый тип данных по флагу **hasIntegers/hasFloats/hasStrings**.
3. Если в ходе обработки встретился хотя бы один элемент соответствующего типа, печатает статистику:

* Краткая (**getShortStats()**) — количество элементов.
* Полная (**getFullStats()**) — дополнительная информация (min, max, sum, average для чисел; min length / max length для строк).

1. Если же соответствующего типа данных не было (например, не встретили ни одного целого), никакой статистики для него не печатается.

**Краткая сводка**

* **process():** главный вход в обработку, вызывает processSingleFile() для каждого входного файла, затем закрывает все потоки и печатает статистику.
* **processSingleFile():** построчно читает файл, вызывает classifyAndWrite() для каждой строки.
* **classifyAndWrite():** определяет тип строки (целое, вещественное, строка) и отправляет в соответствующий метод записи.
* **writeInteger(), writeFloat(), writeString():** при необходимости создают поток записи (через createWriter()), пишут строку, обновляют статистику.
* createWriter(): создаёт BufferedWriter для конкретного файла; учитывает путь **(-o)** и префикс **(-p)**, а также режим дозаписи **(-a)**.
* **closeWriters():** аккуратно закрывает все открытые потоки записи.
* **printStats():** выводит статистику в зависимости от флагов краткой/полной статистики (**-s** или **-f**) и наличия данных каждого типа.

Таким образом, весь процесс фильтрации, записи и сбора статистики зашит именно в этом классе: он полностью контролирует жизненный цикл обработки данных – от чтения файлов до вывода итоговых результатов.