# Инструкция к запуску TestCase.jar

### Требования к среде

Для корректной работы проекта необходимо, чтобы на компьютере была установлена версия Java 21 или выше. Для установки Java можно использовать официальную страницу скачивания (Oracle JDK Downloads) или установить OpenJDK через пакетный менеджер.

### Сборка и запуск проекта

Для сборки проекта TestCase.jar необходимо выполнить следующие действия:

1. **Настройка конфигурации для создания JAR**: чтобы собрать JAR файл, нужно настроить артефакт сборки.
   1. Переходим в **File** > **Project Structure**
   2. В левой части окна выбераем **Artifacts**.
   3. Выбираем **JAR** > **From modules with dependencies**.
2. Настройка артефакта:
   1. В открывшемся окне выбираем основной класс, например класс с методом public static void main(String[] args).
   2. Проверяем, что галочка на **"Include in project build"** установлена.
   3. Выбераем модуль (если в проекте несколько модулей) и путь для вывода JAR файла.
3. Сохраняем настройки.
4. Сборка JAR файла:
   1. Запускаем сборку. После настройки артефакта, чтобы собрать JAR файл, выбираем Build > Build Artifacts > <TestCase>:jar > Build.
   2. Проверяем результат. После завершения сборки JAR файл будет создан в указанной папке. Его также можно найти в проекте в папке out/artifacts или в папке, если она была указана в настройках.
5. Запуск JAR-файла:
   1. Запускаем проект из командной строки. Для этого используем команду: java -jar <путь\_к\_jar\_файлу>/TestCase.jar



Рисунок 1 – Пример запуска программы.

При запуске программы без аргументов, программа указывает на их отсутствие.

1. Параметры запуска проекта

В зависимости от конфигурации проекта и его требований могут быть доступны различные параметры командной строки. Для данной утилиты доступны следующие параметры:

* -s: выводит краткую статистику.
* -f: выводит полную статистику.
* -p <prefix>: указывает префикс для имен выходных файлов.
* -o <path>: указывает путь для сохранения выходных файлов.
* -a: включает режим добавления в существующие файлы, а не перезаписи.

Также после ввода вышеперечисленных параметров программа принимает на вход названия файлов.

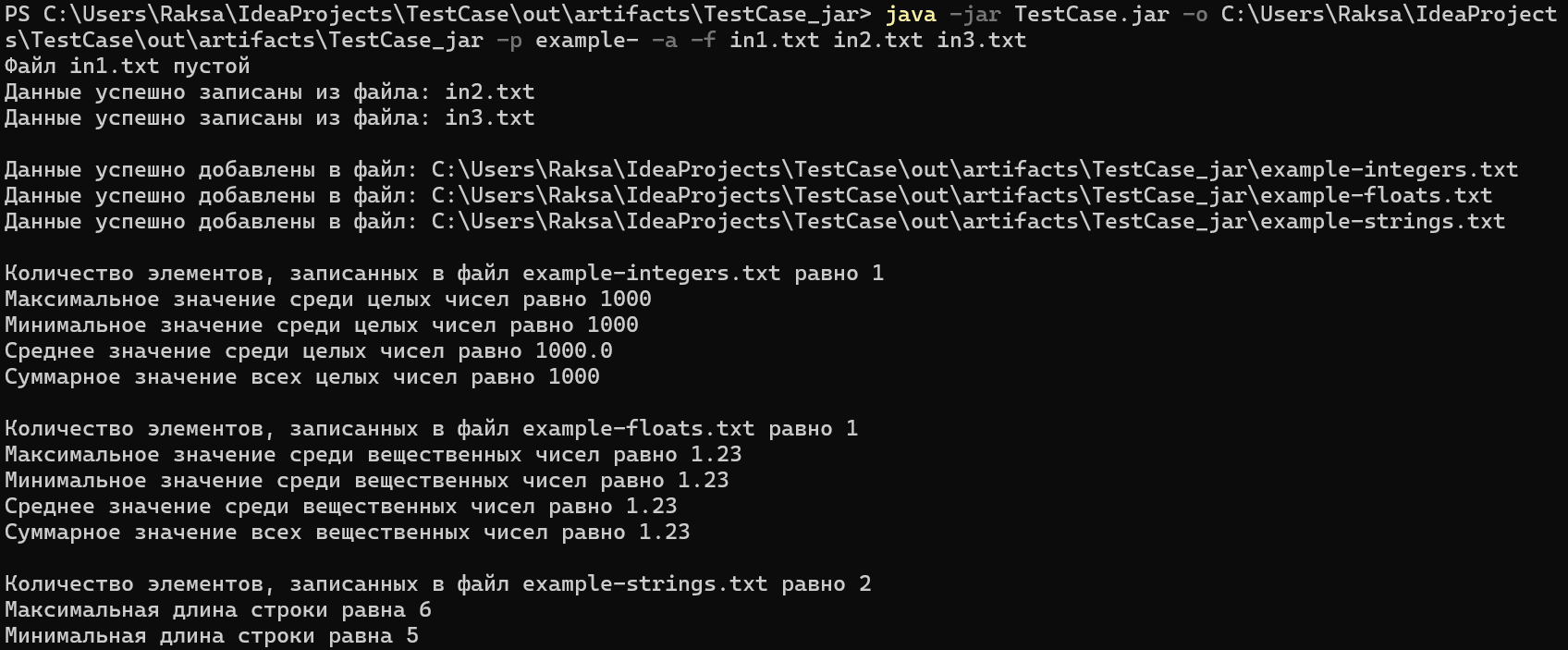


Рисунок 2 – Пример запуска программы с аргументами -f, -p, -o, -a

Как видно на рисунке выше, программа была запущена с указанием конечного пути записи итоговых файлов, префиксом для названия итоговых файлов, параметрами записи в конец файла и выведения полной статистики выполнения данной программы.

1. Особенности реализации программы
   1. Классы

В данном проекте были реализованы 6 классов:

* Main.java – основной класс
* CommandLineParser – класс, отвечающий за обработку входных аргументов командной строки
* FileProc – класс, отвечающий за чтение исходных файлов, обработку в них данных и запись в конечные файлы.
* StringCheck – класс, отвечающий за проверку данных в исходных файлах.
* Statistics – класс, отвечающий за анализ конечных данных.
* Errors – класс, отвечающий за некоторые обрабатываемые исключения
  1. Класс FileProc

В данном классе в методе readFile() дополнительно выводится информация о результатах чтения из исходных файлов: файл пустой, либо информация из него успешно записана.

В методе writeFile(), если указанный путь для финальных файлов отсутствует, то необходимые каталоги будут созданы. При записи данных в финальные файлы программа в зависимости от режима этой записи будет выводить информацию о результате: данные успешно добавлены или перезаписаны в файл.

Кроме того, в этом методе реализовано разделение относительного и абсолютного пути к конечным файлам. Если путь начинается с локального диска (например С:\\), то данный путь будет считаться абсолютным. Если путь начинается с такой последовательности символов «../» (например ../output), то в таком случае путь будет считаться относительным.

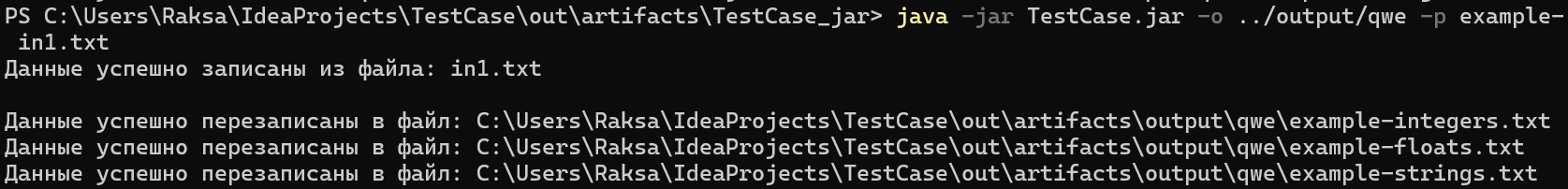


Рисунок 3 – Перезапись конечных файлов

* 1. Класс Statistics

В этом классе для обработки статистики про целочисленные типы данных (метод \_fullStatisticsInt()) используется класс BigInteger, так как он позволяет работать с числами произвольной длины.

* 1. Обрабатываемые исключения

В классе CommandLineParser используются некоторые обрабатываемые исключения, которые содержатся в классе Errors. Рассмотрим их ниже:

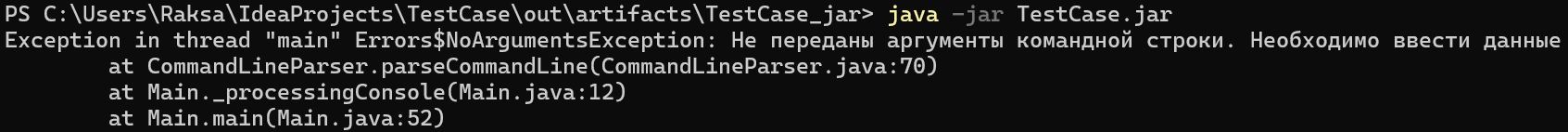


Рисунок 4 – Вызвано исключение, связанное с отсутствием аргументов командной строки

На данном рисунке мы видим, что исключения, которые выбрасываются, добавляются в стек вызовов. В стеке вызовов видно, что оно возникло на уровне метода CommandLineParser.parseCommandLine (строка 70). Далее стек указывает, где этот метод был вызван — в Main.\_processingConsole (строка 12). Затем видно, что \_processingConsole был вызван из метода Main.main (строка 52).

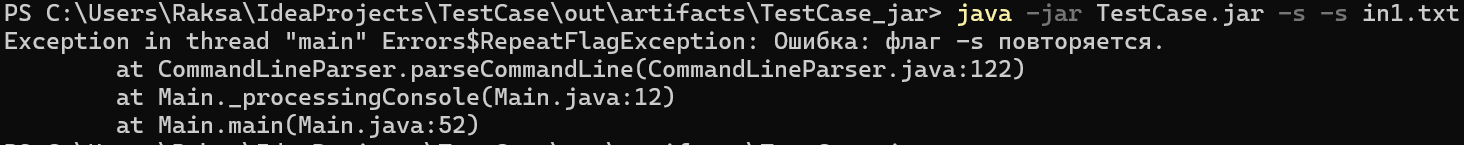


Рисунок 5 – Вызвано исключение, связанное с повторяющимся флагом

Исключение на рисунке 5 будет вызвано, если один из флагов -a, -p, -o, -s, -f повторяется.

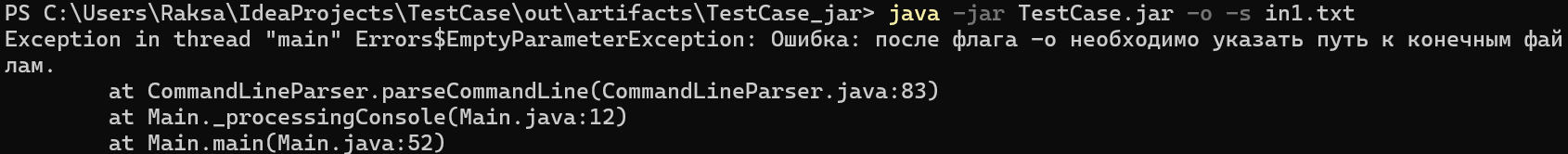


Рисунок 6 – Вызвано исключение, связанное с отсутствием пути после флага -о

Данное исключение будет также вызвано, если после флага -p будет отсутствовать префикс.

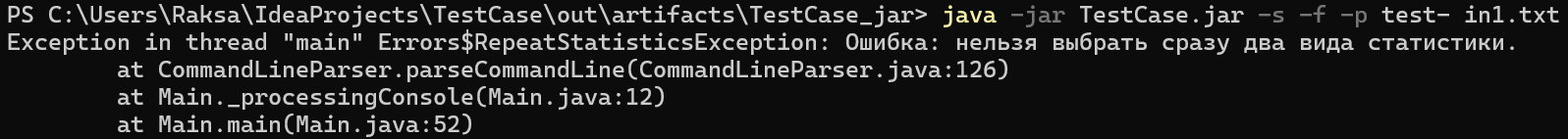


Рисунок 7 – Вызвано исключение, связанное с выбором сразу двух вариантов статистики

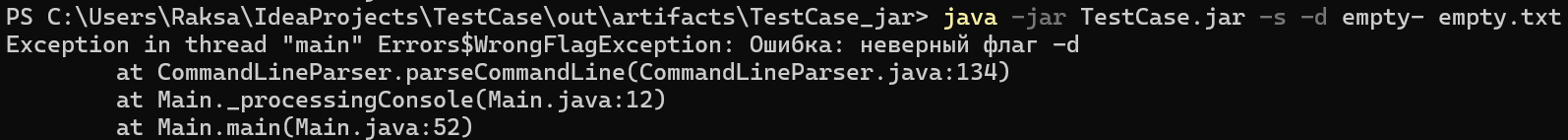


Рисунок 8 – Вызвано исключение, связанное с ошибкой в написании флага

Класс CommandLineParser будет считать флагом любой аргумент, начинающийся на «-».

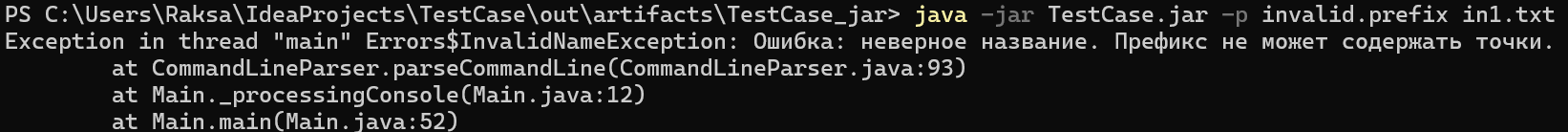


Рисунок 9 – Вызвано исключение, связанное с ошибкой в написании префикса

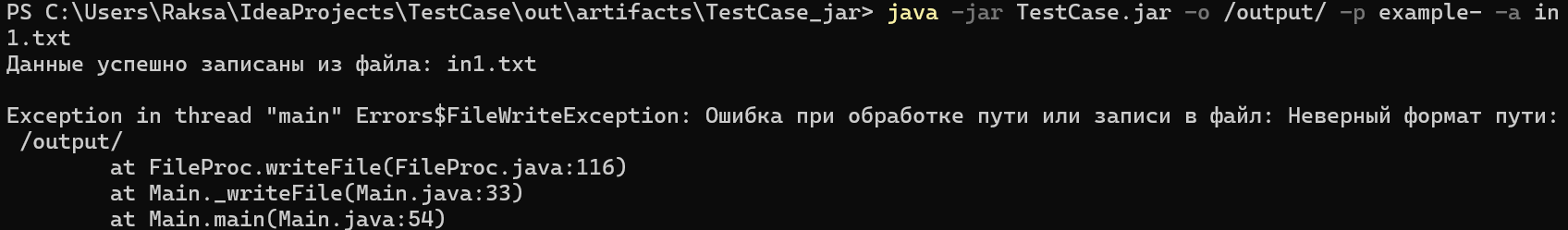


Рисунок 10 – Вызвано исключение, связанное с ошибкой в написании пути к конечным файлам

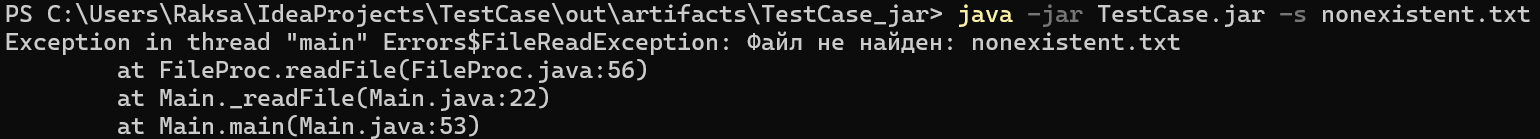


Рисунок 11 – Вызвано исключение, связанное с отсутствием файлов для чтения