**МОЛДАВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет Математики и Информатики**

**Департамент Информатики**

Индивидуальная работа № 1

***Javascript and Typescript***

Проверил: Nartea N.

Выполнил: Iudina Maria grupa IA2304

Кишинев, 2024

Оглавление

[I. Введение. 3](#_Toc163815277)

[II. Практическая часть 4](#_Toc163815278)

[Выводы. 11](#_Toc163815279)

[Библиография 12](#_Toc163815280)

# Введение.

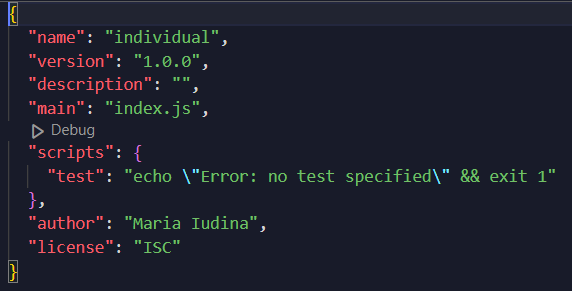
Главной ***целью*** данного проекта является развитие практических навыков создания программного модуля, который позволит пользователям анализировать свои личные финансы на основе предоставленных данных о транзакциях. В рамках этой работы мы ставим перед собой следующие ***цели***:

* ***Подготовка данных:***
* Получение набора данных о транзакциях для анализа.
* Изучение структуры данных и их форматирование для обработки.
* ***Анализ данных:***
* Разработка алгоритмов для анализа транзакций и вычисления статистических показателей (например, общая сумма транзакций, распределение по типам транзакций).
* ***Реализация модуля:***
* Создание программного модуля на языке программирования (например, на JavaScript).
* Реализация функций для обработки и анализа данных о транзакциях.
* ***Тестирование и отладка:***
* Проведение тестирования модуля на различных наборах данных.
* Исправление ошибок и улучшение производительности модуля.
* ***Документирование результатов:***
* Подготовка отчета о выполненной работе, включая описание использованных методов и результатов анализа.
* Презентация результатов работы и полученных выводов.

1. Практическая часть

***Создание файла package.json:***

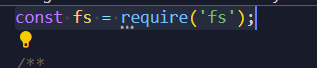
Первым шагом идет инициализация проекта, через команду *npm init*. Эта команда позволяет создать файл *package.json*, в котором идет описание основных настроек проекта, включая его имя, версию, описание и т. д.

**

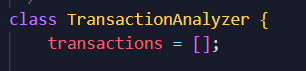
Создание файла *package.json* является важным шагом в начале проекта, который помогает организовать и описать основные настройки и зависимости проекта. Этот файл позволяет другим разработчикам понять цели и особенности проекта, а также управлять им с помощью инструментов *Node.js* и *npm*. Создание хорошо структурированного *package.json* является первым шагом к успешной разработке и документации индивидуального проекта.

Далее мы переходим к самой работе, первым делом подключая файл, из которого мы будем брать всю необходимую информацию. В данном случае, это файл со всеми транзакциями, которые мы будем анализировать в ходе нашей работы:

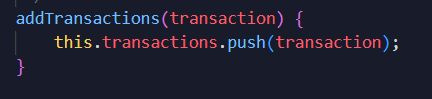
Код const fs = require('fs'); в JavaScript используется для подключения модуля *fs (File System)* из стандартной библиотеки *Node.js.* Этот модуль предоставляет функции для работы с файловой системой компьютера, такие как чтение файлов, запись файлов, создание директорий и другие операции ввода-вывода.



Мы определили класс *TransactionAnalyzer*, который содержит свойство *transactions* для хранения массива транзакций. Конструктор класса принимает массив транзакций (по умолчанию пустой).



Затем мы создаем экземпляр класса *TransactionAnalyzer*, передавая ему массив с транзакциями, добавляем новую транзакцию с помощью *addTransaction*, а затем вызываем различные методы для анализа данных о транзакциях.

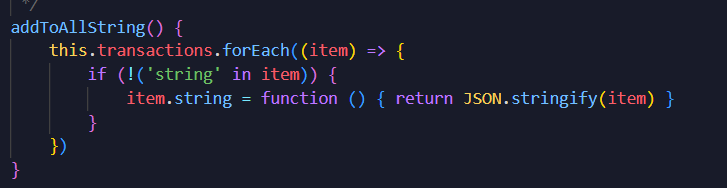


Этот пример демонстрирует использование класса *TransactionAnalyzer* для анализа данных о финансовых транзакциях.

В данной работе есть различные ***методы***, которые выполняют разного рода анализ транзакций, таких как:

1. Возвращает массив всевозможных типов транзакций (например, ['debit', 'credit']).
2. Рассчитывает общую сумму всех транзакций.
3. Вычисляет общую сумму транзакций за указанный год, месяц и день.
4. Возвращает транзакции указанного типа (debit или credit).
5. Возвращает транзакции, проведенные в указанном диапазоне дат от начала до конца (имеется в виду даты).
6. Возвращает транзакции, совершенные с указанным торговым местом или компанией.
7. Возвращает среднее значение транзакций.
8. Возвращает транзакции с суммой в заданном диапазоне от minAmount до maxAmount.
9. Вычисляет общую сумму дебетовых транзакций.
10. Возвращает месяц, в котором было больше всего транзакций.
11. Возвращает месяц, в котором было больше дебетовых транзакций.
12. Возвращает каких транзакций больше всего.
13. Возвращает debit, если дебетовых.
14. Возвращает credit, если кредитовых.
15. Возвращает equal, если количество равно.
16. Возвращает транзакции, совершенные до указанной даты.
17. Возвращает транзакцию по ее уникальному идентификатору.
18. Возвращает новый массив, содержащий только описания транзакций.

В данном отчете мы будем анализировать ключевые моменты, так как некоторые методы используют функции, которые похожи между собой или выполняют примерно одни и те же действия.



***Разбор метода addToAllString():***

*this.transactions.forEach((item) => { ... })*: Этот код использует метод forEach(), чтобы выполнить итерацию по каждому элементу в массиве transactions.

*if (!('string' in item)) { ... }:* Здесь проверяется, отсутствует ли у текущего элемента свойство string. Оператор in проверяет наличие свойства в объекте.

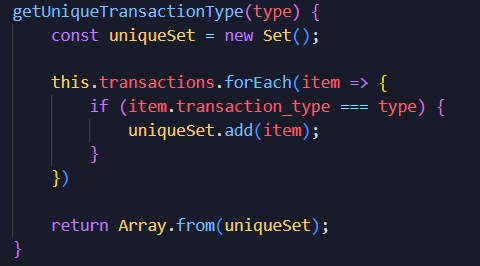
*item.string = function () { return JSON.stringify(item) }*: Если у элемента отсутствует свойство string, то создается новое свойство string, которое является функцией. Эта функция использует JSON.stringify() для преобразования текущего элемента item в строку JSON.

***В этом примере:***

Создается экземпляр класса *TransactionAnalyzer*, и метод *addToAllString()* вызывается для добавления функции *string* ко всем элементам массива *transactions*.

После этого каждая транзакция из массива *transactions* содержит метод *string()*, который преобразует транзакцию в строку *JSON* при вызове.

При вызове *transaction.string()* для каждой транзакции выводится строковое представление этой транзакции с помощью *JSON.stringify().*



***Разбор метода getUniqueTransactionType(type):***

*const uniqueSet = new Set();:* Создается новый объект *Set* под названием *uniqueSet*, который будет содержать уникальные элементы (транзакции).

*this.transactions.forEach(item => { ... }):* Здесь используется метод *forEach()* для итерации по массиву *transactions*. Для каждой транзакции (*item*) проверяется условие *item.transaction\_type === type.*

*if (item.transaction\_type === type) { uniqueSet.add(item); }*: Если тип транзакции *item* соответствует заданному типу *type*, то данная транзакция добавляется в множество *uniqueSet* с помощью метода *add().* Объект *Set* гарантирует, что каждый элемент в нем будет уникальным.

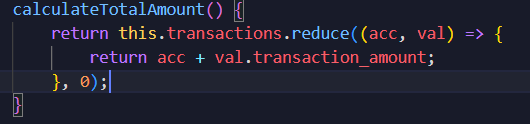
*return Array.from(uniqueSet);:* После завершения итерации по массиву *transactions*, объект *uniqueSet* преобразуется в массив с помощью *Array.from(uniqueSet)*, и этот массив уникальных транзакций возвращается из метода.

***В этом примере:***

Создается экземпляр класса *TransactionAnalyzer* с массивом *transactions*.

Вызывается метод *getUniqueTransactionType('debit'),* который возвращает массив уникальных транзакций с типом *'debit'*.

Аналогично вызывается метод *getUniqueTransactionType('credit')*, который возвращает массив уникальных транзакций с типом *'credit'*.



***Разбор метода calculateTotalAmount():***

*this.transactions.reduce((acc, val) => { ... }, 0):* Здесь используется метод *reduce()* для выполнения агрегации (свертки) массива *transactions* в одно значение. Первым параметром передается функция обратного вызова *(acc, val) => { ... }*, которая будет применяться к каждому элементу массива.

*acc* (аккумулятор) - это накопленное значение (сумма), которое передается между итерациями.

*val* (значение) - текущий элемент массива, к которому применяется функция.

*return acc + val.transaction\_amount;:* В каждой итерации функция прибавляет значение *val.transaction\_amount* (сумма транзакции) к текущему аккумулятору *acc*.

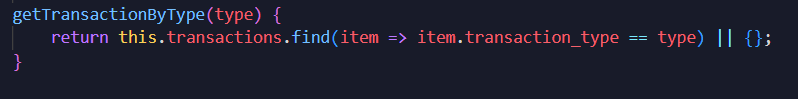
*}, 0):* Начальное значение аккумулятора (*acc*) установлено в 0, что означает, что сумма начинается с нуля.

***В этом примере:***

Создается экземпляр класса *TransactionAnalyzer* с массивом *transactions*.

Вызывается метод *calculateTotalAmount()*, который возвращает общую сумму всех транзакций из массива.

Общая сумма транзакций сохраняется в переменной *totalAmount* и выводится в консоль.



***Разбор метода getTransactionByType(type):***

*this.transactions.find(item => item.transaction\_type == type):* Этот код использует метод *find()* для поиска первой транзакции (*item*), у которой значение свойства *transaction\_type* равно указанному *type*.

*item => item.transaction\_type == type:* Это функция обратного вызова, которая принимает каждый элемент (*item*) из массива *transactions* и возвращает *true*, если *transaction\_type* удовлетворяет условию *item.transaction\_type == type*, то есть тип транзакции совпадает с указанным *type*.

*|| {}:* Если метод *find()* не находит ни одной транзакции с указанным типом *type*, он вернет *undefined*. Чтобы избежать ошибки, связанной с попыткой обращения к свойствам *undefined*, используется оператор ||, который возвращает пустой объект {} в случае отсутствия найденной транзакции.

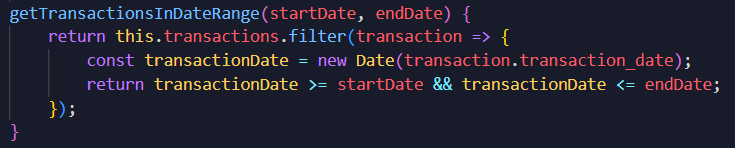
***В этом примере:***

Создается экземпляр класса *TransactionAnalyzer* с массивом *transactions*.

Вызывается метод *getTransactionByType('debit'),* чтобы найти первую транзакцию с типом *'debit'*.

Аналогично вызывается метод *getTransactionByType('credit'),* чтобы найти первую транзакцию с типом *'credit'*.

Также вызывается метод *getTransactionByType('unknown'),* чтобы проверить, что возвращается пустой объект при отсутствии нужной транзакции.



***Разбор метода getTransactionsInDateRange(startDate, endDate):***

*this.transactions.filter(transaction => { ... }):* Этот код использует метод *filter()*, чтобы вернуть новый массив, содержащий только те элементы (транзакции), для которых функция обратного вызова возвращает *true*.

*const transactionDate = new Date(transaction.transaction\_date);:* Здесь для каждой транзакции создается объект *Date*, преобразуя строку *transaction.transaction\_date* в дату. Предполагается, что *transaction.transaction\_date* представляет собой строку в формате, который может быть преобразован в объект *Date*.

*return transactionDate >= startDate && transactionDate <= endDate;:* Возвращается *true* для транзакции, если дата этой транзакции находится в указанном диапазоне между *startDate* и *endDate*.

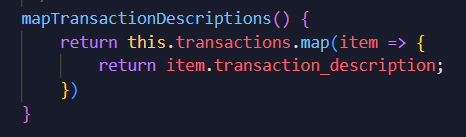
***В этом примере:***

Создается экземпляр класса *TransactionAnalyzer* с массивом *transactions*.

Устанавливаются *startDate* и *endDate* как объекты *Date* для указания начальной и конечной даты диапазона.

Вызывается метод *getTransactionsInDateRange(startDate, endDate),* который фильтрует массив *transactions* и возвращает только те транзакции, даты которых находятся в заданном диапазоне между *startDate* и *endDate*.

Отфильтрованный массив транзакций выводится в консоль.



***Разбор метода mapTransactionDescriptions():***

*this.transactions.map(item => { ... }):* Этот код использует метод *map(),* чтобы создать новый массив, содержащий результаты вызова функции обратного вызова для каждого элемента массива *transactions*.

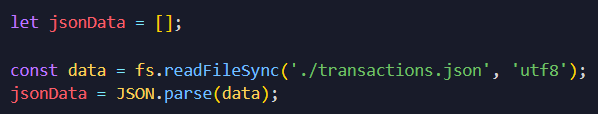
*item => item.transaction\_description:* Для каждой транзакции (*item*) из массива *transactions* возвращается значение свойства *transaction\_description.* Таким образом, новый массив будет содержать только описания транзакций.

***В этом примере:***

Создается экземпляр класса *TransactionAnalyzer* с массивом *transactions*.

Вызывается метод *mapTransactionDescriptions(),* который преобразует массив *transactions* в новый массив, содержащий только описания транзакций *(transaction\_description).*

Результат выполнения метода выводится в консоль.



***В этом примере:***

Импортируется модуль *fs (File System)* для работы с файловой системой *Node.js*.

Функция *readFileSync('./transactions.json', 'utf8')* считывает содержимое файла *transactions.json* как текст в кодировке *UTF-8*.

Результат чтения файла сохраняется в переменной *data*.

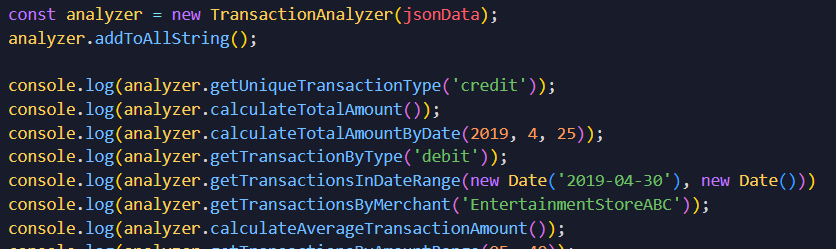
*JSON.parse(data)* преобразует содержимое файла из формата *JSON* в объект *JavaScript*.

Результат парсинга *JSON* сохраняется в переменную *jsonData*.

В случае успешной загрузки и парсинга *JSON* данные выводятся в консоль.

Обрабатывается возможная ошибка при чтении или парсинге файла.

Этот подход считывания данных из файла и их преобразования из формата *JSON* в объекты *JavaScript* позволяет использовать данные из внешнего файла в вашем приложении для дальнейшей обработки или анализа.



***В этом примере:***

*jsonData* загружается из файла *transactions.json.*

Создается экземпляр *analyzer* класса *TransactionAnalyzer* с использованием *jsonData*.

Вызываются методы, для анализа и сортировки наших транзакций и выводятся на экран пользователю.

Этот код позволяет анализировать данные транзакций, применять к ним определенные операции и получать результаты анализа, такие как уникальные типы транзакций и общая сумма транзакций, которые можно использовать дальше в вашем приложении или для составления отчетов.

# Выводы.

При разработке класса TransactionAnalyzer для анализа данных транзакций в JavaScript был создан набор методов и функций, позволяющих эффективно обрабатывать массив транзакций и извлекать полезную информацию.

Класс TransactionAnalyzer представляет собой инструмент для анализа данных транзакций.

Класс TransactionAnalyzer предоставляет мощный инструмент для анализа данных транзакций. Его методы позволяют выполнять различные операции, такие как фильтрация, суммирование, поиск и вычисления на основе свойств транзакций. Этот класс может быть полезен для структурирования и анализа данных финансовых транзакций, что является важной задачей во многих приложениях, связанных с финансами или учетом. Каждый метод обеспечивает определенную функциональность, позволяя эффективно извлекать и обрабатывать информацию о транзакциях с помощью простых вызовов функций.

# Библиография

1. Материалы лабораторной №1 на сайт курса (<https://github.com/MSU-Courses/javascript_typescript/blob/main/lab/lab_guidelines.md>).

2. <https://learn.javascript.ru/first-steps>

3. <https://developer.mozilla.org/ru/docs/Web/JavaScript>

4. <https://devdocs.io/javascript/>