МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ

Національний університет «Одеська юридична академія»

Факультет кібербезпеки та інформаційних технологій

Кафедра інформаційних технологій

Протокол практичної роботи №8

з дисципліни об'єктно-орієнтоване програмування

на тему: «Колекції об’єктів»

Виконав студент групи

ІПЗ-212

Корнійчук М. М.

Прийняв

Рудніченко М. Д.

Одеса, 2022

ЗМІСТ

[ВСТУП](#_ki7i591yqske) 3

[ТЕОРЕТИЧНА ЧАСТИНА](#_335513nxkbjl) 4

[ПРАКТИЧНА ЧАСТИНА](#_9cm35ikas1ko) 4

[ВИСНОВОК](#_xlxs31hqwy0n) 19

[ЛІТЕРАТУРА](#_5m79jhxx931e) 20

# 

# ВСТУП

Мета роботи:

* Ознайомитися з поняттям колекції та відмінністю колекції від структур даних;
* Вивчити різні типи колекцій;
* Навчитися використовувати колекції для вирішення поставлених задач.

# ТЕОРЕТИЧНА ЧАСТИНА

В Java имеется несколько способов хранения объектов (или, точнее, ссылок на объекты). Самый простой вариант хранения объектов – массивы. Массив обеспечивает самый эффективный способ хранения групп объектов. Однако массив имеет фиксированный размер, а в общем случае во время написания программы разработчик может не знать точное количество объектов или необходимо более эффективные способы хранения объектов. К тому же, массивы неэффективны при большом объеме элементов. Для таких случаев, в Java и других языках реализован механизм коллекций объектов.

**Коллекции** – это хранилища, поддерживающие различные способы накопления и упорядочения объектов с целью обеспечения возможностей эффективного доступа к ним. Они представляют собой реализацию абстрактных типов (структур) данных, поддерживающих три основные операции:

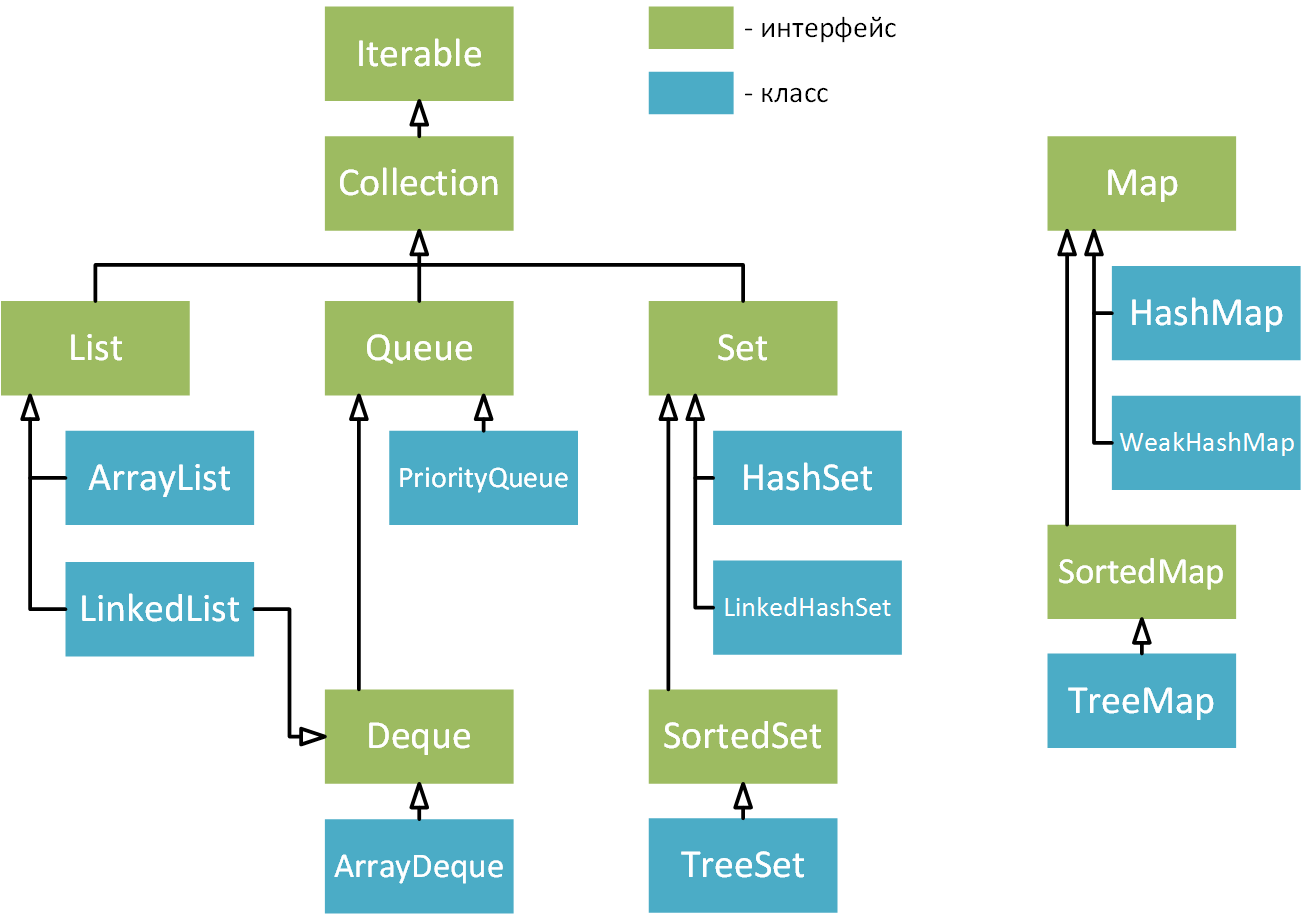
* добавление нового элемента в коллекцию;
* удаление элемента из коллекции;
* изменение элемента в коллекции.

Коллекции в Java объединены в библиотеке классов **java.util** и представляют собой контейнеры для хранения и манипулирования объектами.

Основные интерфейсы коллекций:

* **Map<K,V>** – карта отображения вида «ключ-значение»;
* **Collection<E>** – вершина иерархии остальных коллекций;
* **List<E>** – специализирует коллекции для обработки списков;
* **Set<E>** – специализирует коллекции для обработки множеств, содержащих уникальные элементы.

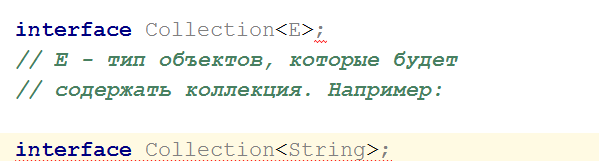
На изображении ниже приведены основные интерфейсы и классы коллекций



*Основные интерфейсы и классы коллекций в Java*

**ИНТЕРФЕЙС COLLECTION**

Этот интерфейс служит основанием, на котором построен весь каркас коллекций, поскольку он должен быть реализован почти всеми классами коллекций (кроме коллекций, реализующих интерфейс **Map**). Интерфейс Collection является обобщенным и объявляется следующим образом:



Интерфейс **Collection** расширяет интерфейс **Iterable** (interface Collection extends Iterable). Это означает, что все коллекции можно перебирать, организовав цикл **for** в стиле **for each**. В интерфейсе Collection определяются основные методы, которые должны иметь все коллекции. Некоторые основные методы перечислены в таблице ниже.

*Некоторые основные методы, определенные в интерфейсе* ***Collection***

|  |  |
| --- | --- |
| Метод | Описание |
| add(Object o) | Добавляет указанный объект в коллекцию |
| remove(Object o) | Удаляет указанный объект из коллекции |
| clear() | Удаляет все элементы из коллекции |
| size() | Возвращает количество элементов в коллекции |
| iterator() | Возвращает объект, который используется для доступа к элементам коллекции |

**Интерфейс List**

Список (**List**) представляет собой упорядоченный набор элементов и может содержать повторяющиеся элементы. Вы можете получить доступ к любому элементу по индексу. Список представляет собой динамический массив. Список является одним из наиболее используемых типов коллекций.

Основные свойства коллекций, реализующих интерфейс **List**:

* список может включать одинаковые элементы;
* элементы в списке хранятся в том порядке, в котором они помещались;
* можно получить доступ к любому элементу по его порядковому номеру (индексу) внутри списка

Существуют две основные разновидности **List**:

1) Базовый контейнер **ArrayList** с превосходной скоростью произвольного доступа к элементам, но относительно медленными операциями вставки и удаления элементов в середине. Пожалуй, самая часто используемая коллекция. **ArrayList** инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.

Так как **ArrayList** использует массив, то  время доступа к элементу по индексу минимально (в отличие от **LinkedList**). При удалении произвольного элемента из списка, все элементы находящиеся «правее» смещаются на одну ячейку влево, при этом реальный размер массива (его емкость, **capacity**) не изменяется. Если при добавлении элемента, оказывается, что массив полностью заполнен, будет создан новый массив размером (n \* 3) / 2 + 1, в него будут помещены все элементы из старого массива плюс новый, добавляемый элемент.

2) Связанный список **LinkedList** с оптимальным последовательным доступом и низкозатратными операциями вставки и удаления в середине списка. Операции произвольного доступа **LinkedList** выполняет относительно медленно, но обладает более широкой функциональностью, чем **ArrayList**.

Это структура данных, состоящая из узлов, каждый из которых содержит как собственно данные, так и две ссылки («связки») на следующий и предыдущий узел списка. Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время — ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, **LinkedList** в абсолютных величинах проигрывает **ArrayList** и по потребляемой памяти и по скорости выполнения операций.

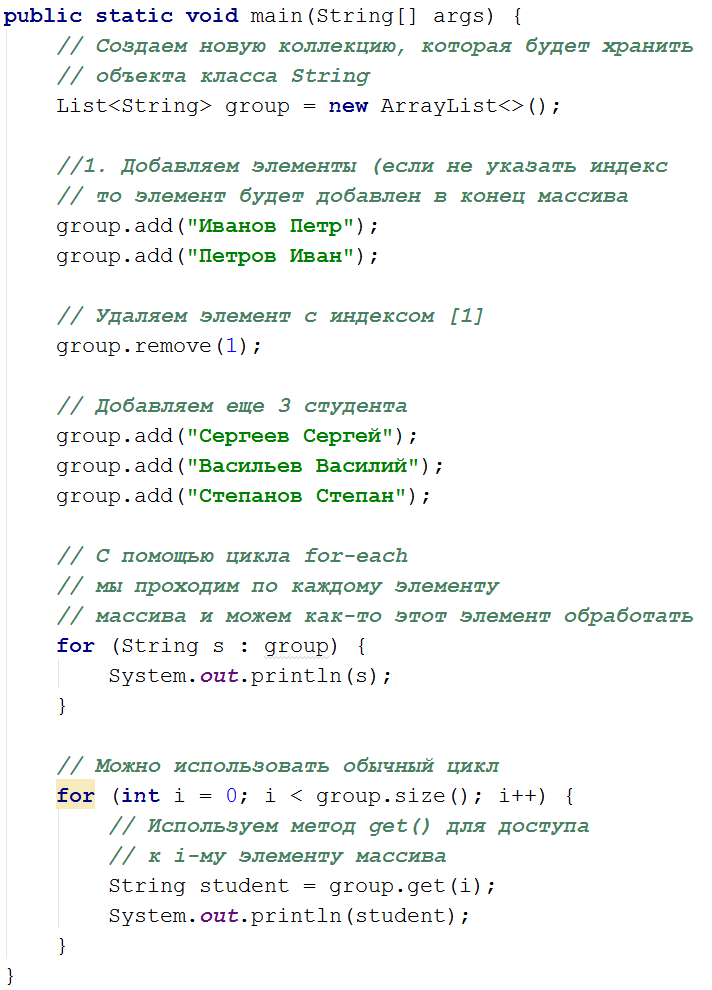
**В общем случае, следует использовать ArrayList. Коллекцию LinkedList имеет смысл использовать в случае, если происходит интенсивная вставка\удаление в середину списка либо необходимо гарантированное, заранее известное время добавления элемента в список.**

В таблице ниже приведены некоторые методы интерфейса **List**, которые используются в **ArrayList** и **LinkedList**.

*Некоторые основные методы, определенные в интерфейсе* ***List***

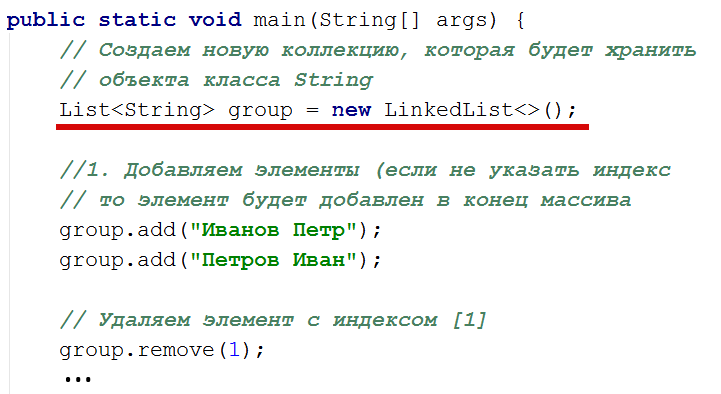
|  |  |
| --- | --- |
| Метод | Описание |
| void add(int index, E obj) | Добавляет в список по индексу *index* объект *obj* |
| boolean addAll(int index, Collection<? extends E> col) | добавляет в список по индексу *index* все элементы коллекции *col*. Если в результате добавления список был изменен, то возвращается *true*, иначе возвращается *false*. |
| E get(int index) | Возвращает объект из списка по индексу *index* |
| int indexOf(Object obj) | Возвращает индекс первого экземпляра заданного объекта в вызывающем списке. Если заданный объект отсутствует в списке, возвращается значение -1 |
| E remove(int index) | Удаляет объект из списка по индексу *index*, возвращая при этом удаленный объект |
| E set(int index, E obj) | Присваивает значение объекта *obj* элементу, который находится по индексу *index* (фактически, записывает новую ссылку на объект поверх старой, метод возвращает старую ссылку) |
| List<E> sublist(int start, int end) | Получает набор элементов, которые находятся в списке между индексами *start* и *end* |

Рассмотрим пример работы с коллекциями **ArrayList** и **LinkedList**



Обратите внимание: тип **ArrayList** преобразуется в **List** посредством восходящего преобразования. В идеале большая часть кода должна взаимодействовать с этими интерфейсами (хотя это не всегда возможно), а точный тип указывается только в точке создания контейнера.

Если вы вдруг решитесь изменить реализацию интерфейса, для этого достаточно внести изменение в точке создания



**Интерфейс Queue**

Этот интерфейс расширяет интерфейс **Collection** и определяет поведение очереди, которая действует как список по принципу «первым вошел – первый вышел». Иначе говоря, объекты помещаются в один «конец» очереди, а извлекаются с другого «конца». Таким образом, порядок занесения объектов в контейнер будет совпадать с порядком их извлечения оттуда. Соответственно, очереди часто используются для надежного перемещения объектов из одной области программы в другую. Очереди также играют важную роль в параллельном программировании, потому что они обеспечивают безопасную передачу объектов между задачами.

Существует разные виды очередей, порядок организации в которых основывается на некотором критерии. Интерфейс **Queue** является обобщенным и объявляется следующим образом:

interface Queue<E>

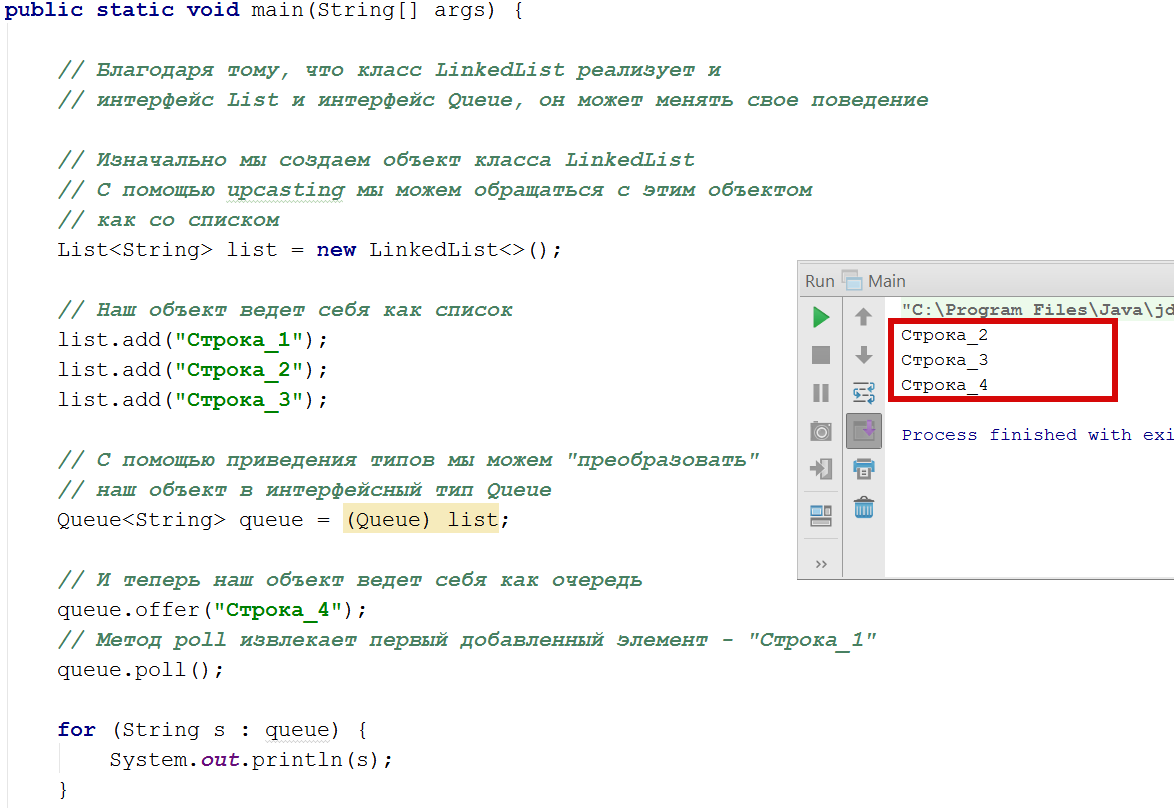
где **E** обозначает тип объектов, которые будут храниться в очереди. Ниже перечислены основные методы, определенные в интерфейсе **Queue**.

*Некоторые основные методы, определенные в интерфейсе* ***Queue***

|  |  |
| --- | --- |
| Метод | Описание |
| E element() | Возвращает элемент из головы очереди. Возвращаемый элемент не удаляется. Если очередь пуста, генерируется исключение типа **NoSuchElementException** |
| boolean offer(E object) | Пытается ввести заданный *object* в очередь. Возвращает логическое значение *true*, если *object* введен, иначе – *false* |
| E peek() | Возвращает элемент из головы очереди. Если очередь пуста, возвращает пустое значение *null*. Возвращаемый элемент не удаляется из очереди |
| E poll() | Возвращает элемент из головы очереди и удаляет его. Если очередь пуста, возвращает пустое значение *null* |
| E remove() | Удаляет элемент из головы очереди, возвращая его. Генерирует исключение типа **NoSuchElementException**, если очередь пуста |

**Класс LinkedList**

Класс **LinkedList**, который мы рассматривали в связи с интерфейсом **List**, также реализует интерфейс **Deque** и поддерживает поведение очередей. Восходящее преобразование **LinkedList** в **Queue** позволяет использовать объект как очередь

**Класс PriorityQueue**

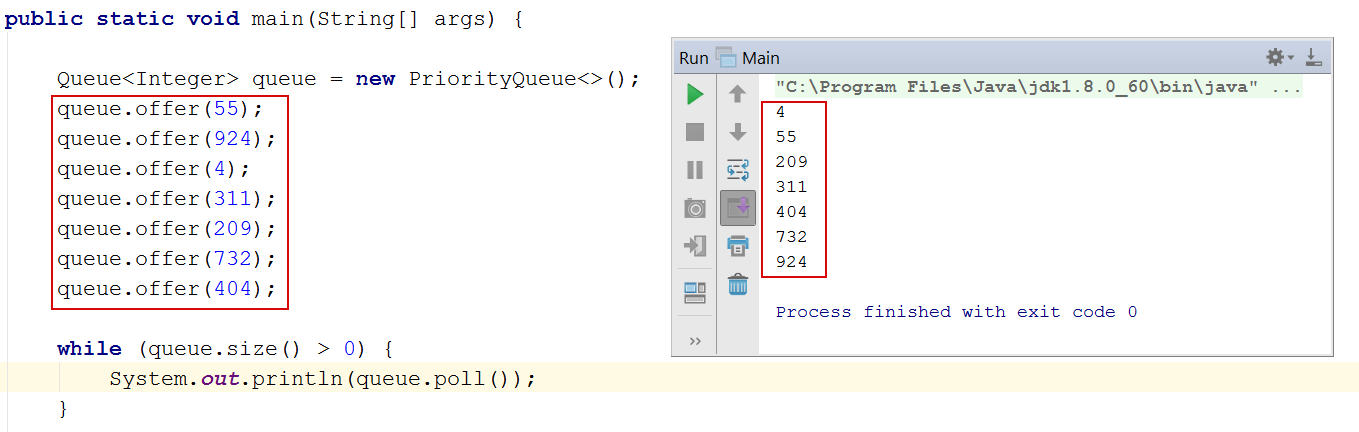
Класс PriorityQueue расширяет класс AbstractQueue и реализует интерфейс Queue. Он служит для создания очереди по приоритетам на основании компаратора очереди.

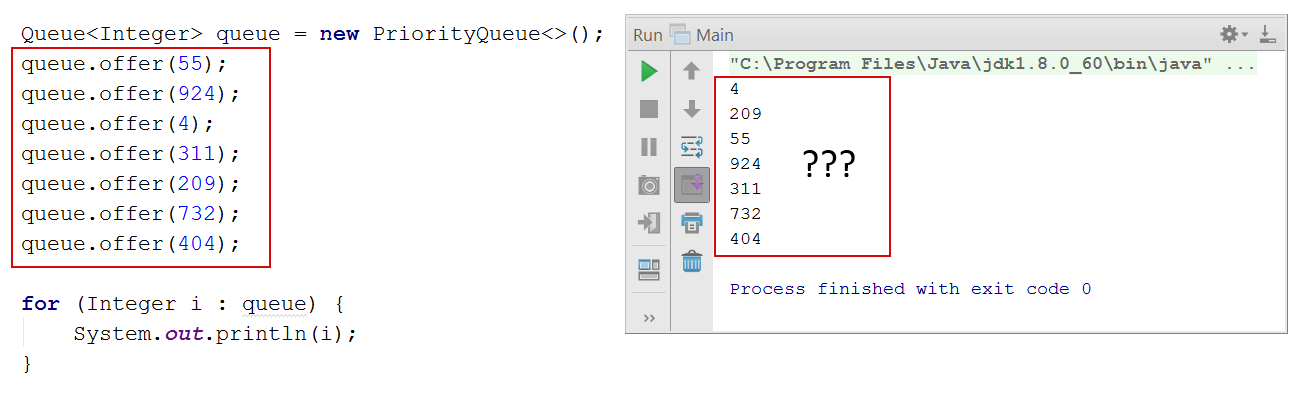
В приоритетной очереди следующим извлекается элемент, обладающий наивысшим *приоритетом*. Например, в аэропорту клиент может быть обслужен вне очереди, если его самолет готовится к вылету. В системе передачи сообщений некоторые сообщения могут содержать более важную информацию; они должны быть срочно обработаны независимо от времени поступления.

Класс **PriorityQueue** является обобщенным и объявляется следующим образом:

class PriorityQueue<E>

Если при построении очереди компаратор не указан, то применяется компаратор, выбираемый по умолчанию для того типа данных, который сохраняется в очереди. Таким образом, в начале (голове) очереди окажется элемент с наименьшим значением. Но, предоставляя свой компаратор, можно задать другую схему сортировки элементов в очереди. Например, когда в очереди сохраняются элементы, содержащие метку времени, для этой очереди можно задать приоритеты таким образом, чтобы самые давние элементы располагались в начале очереди.

**ВНИМАНИЕ!** Класс **PriorityQueue** имеет одну примечательную особенность. Так как очередь с приоритетами реализована с помощью кучи (бинарное дерево с определенными свойствами), то попытка вывестиэлементы с помощью цикла **foreach**, скорее всего, приведет к неправильному выводу элементов.

**Интерфейс Set**

В интерфейсе **Set** определяется множество. Он расширяет интерфейс **Collection** и определяет поведение коллекций, не допускающих дублирования элементов. Таким образом, метод **add()** возвращает логическое значение **false** при попытке ввести в множество дублирующий элемент. В этом интерфейсе не определяется никаких дополнительных методов. Интерфейс Set является обобщенным и объявляется приведенным ниже образом, где E обозначает тип объектов, которые должно содержать множество

interface Set<E>

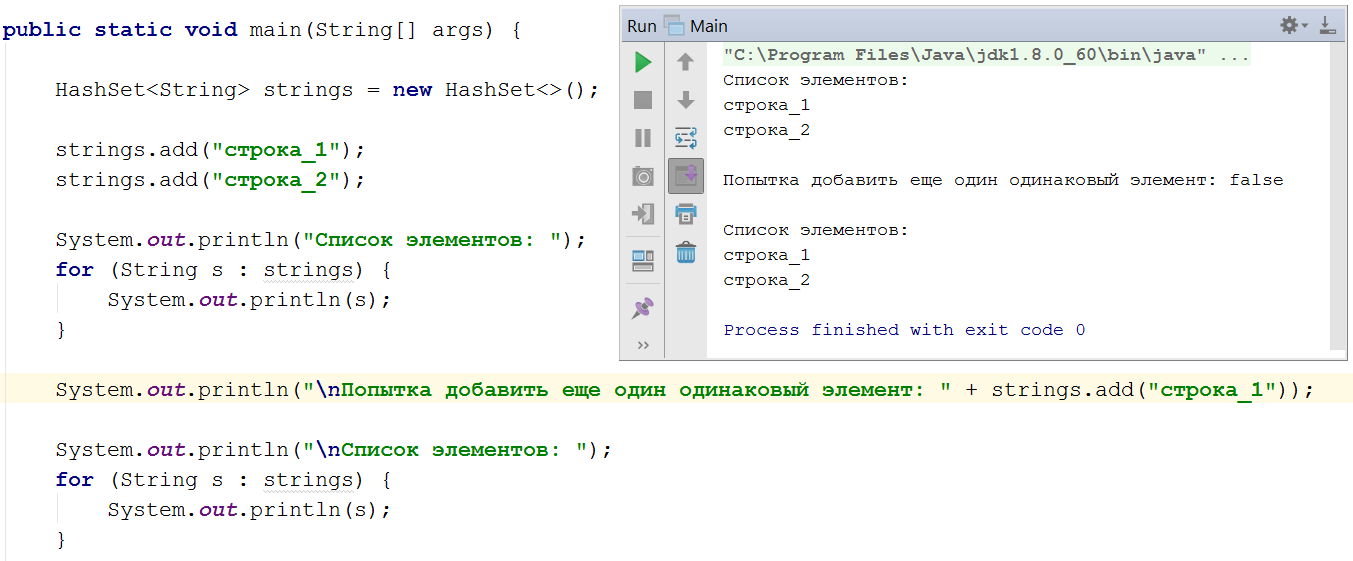
Как уже было сказано, Set обладает таким же интерфейсом, как и Collection, поэтом в Set нет дополнительной функциональности. Вместо этого Set представляет собой разновидность Collection – просто обладает другим поведением.

**Класс HashSet**

Хеш-таблица хранит информацию, используя так называемый механизм хеширования, в котором содержимое ключа используется для определения уникального значения, называемое хеш-кодом. Этот хеш-код затем применяется в качестве индекса, с которым ассоциируются данные, доступные по этому ключу. Преобразование ключа в хеш-код выполняется автоматически – вы никогда не узнаете самого хеш-кода. Также ваш код не может напрямую индексировать хеш-таблицу.

**Выгода от хеширования состоит в том, что оно обеспечивает константное время выполнения методов add(), contains(), remove() и size(), даже для больших объемов данных.**

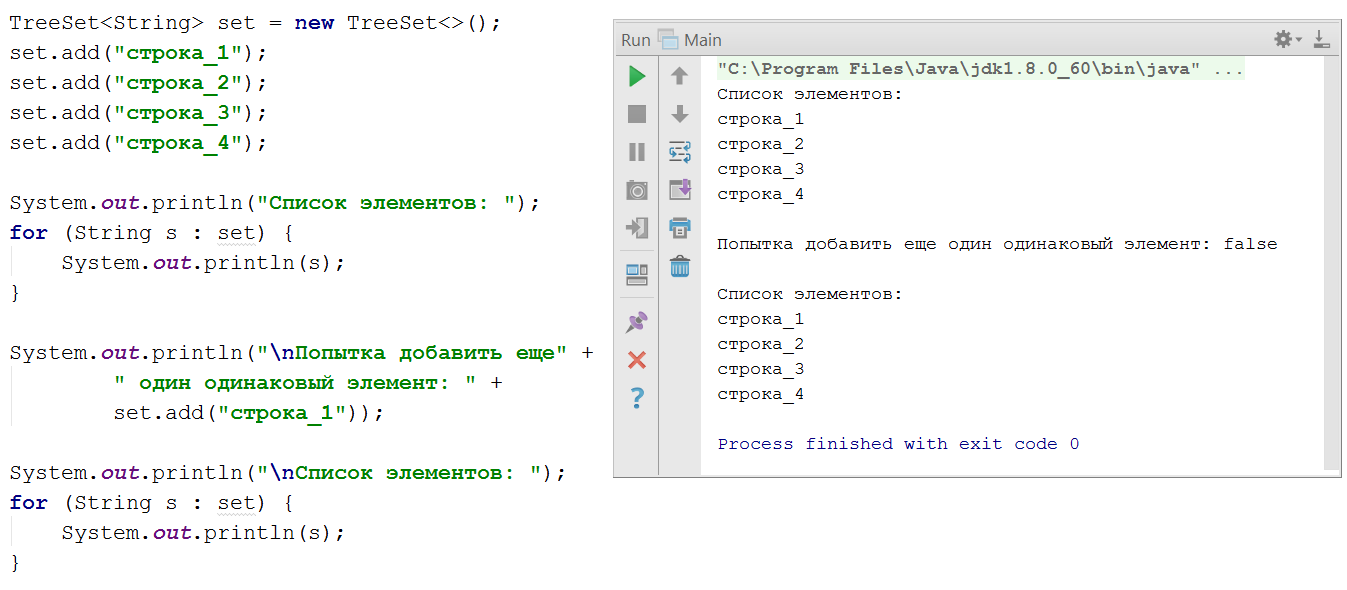
Если вы хотите использовать **HashSet** для хранения объектов собственных пользовательских классов, то вы ДОЛЖНЫ переопределить методы **hashCode()** и **equals()**, иначе два логически-одинаковых объекта будут считаться разными, так как при добавлении элемента в коллекцию будет вызываться метод **hashCode()** класса **Object** (который, скорее всего, вернет разный хеш-код для двух, логически одинаковых объектов).



Важно отметить, что класс **HashSet** не гарантирует упорядоченности элементов, поскольку процесс хеширования сам по себе обычно не порождает сортированных наборов. Если вам нужны сортированные наборы, то лучшим выбором может быть класс **TreeSet**.

**Класс TreeSet**

**TreeSet** – коллекция, которая хранит свои элементы в виде упорядоченного по значениям дерева. **TreeSet** использует сбалансированное красно-черное дерево для хранения элементов. Эта коллекция хороша тем, что для операций **add()**, **remove()**, **contains()** потребуется гарантированное время **log(n)**.



**ИНТЕРФЕЙС MAP**

Интерфейс **Map** описывает коллекцию, состоящую из пар «ключ – значение». У каждого ключа только одно значение, что соответствует математическому понятию однозначной функции или отображения. Такую коллекцию часто называют еще словарем (**dictionary**) или ассоциативным массивом (**associative array**). Несмотря на то, что интерфейс **Map** входит в список коллекций **Java**, он не расширяет интерфейс **Collection**.

Интерфейс **Map** соотносит уникальные ключи со значениями. Ключ – это объект, который вы используете для последующего извлечения данных. Задавая ключ и значение, вы можете помещать значения в объект **Map**. После того как это значение сохранено, вы можете получить его по ключу. Интерфейс Map – это обобщенный интерфейс, объявленный так, как показано ниже:

interface Map<K, V>

Здесь **K** указывается тип ключей, а **V** – тип хранимых значений. Ниже перечислены основные методы, определенные в интерфейсе **Map**.

*Некоторые основные методы, определенные в интерфейсе* ***Map***

|  |  |
| --- | --- |
| Метод | Описание |
| boolean containsValue(Object v) | Возвращает **true**, если коллекция содержит значение **v** |
| boolean containsKey(Object k) | Возвращает **true**, если коллекция содержит ключ **k** |
| V get(Object k) | Возвращает значение объекта, ключ которого равен **k**. Если такого элемента не окажется, то возвращается значение **null** |
| V put (K k, V v) | Помещает в коллекцию новый объект с ключом **k** и значением **v**. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа **k**, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается **null** |
| V remove(Object k) | Удаляет объект с ключом **k** |
| Set<Map.Entry<K,V>> entrySet() | Возвращает набор элементов коллекции. Все элементы представляют объект **Map.Entry** |

Обращение с отображениями опирается на две основные операции, выполняемые методами **get()** и **put()**. Чтобы ввести значение в отображение, следует вызвать **put()**, указав ключ и значение, а для того чтобы получить значение из отображения – вызвать метод **get()**, передав ему ключ в качестве аргумента. По этому ключу будет возвращено связанное с ним значение.

Как упоминалось ранее, **Map** не реализует интерфейс **Collection**, хотя являются частью каркаса коллекций. Тем не менее, можно получить представление отображения в виде коллекции. Для этого можно воспользоваться методом **entrySet()**, возвращающим множество, содержащее элементы отображения (для ключей используйте метод **keySet()**, а для значений – **values()**).

**Интерфейс Map.Entry**

Этот интерфейс позволяет обращаться с отдельными записями в отображении. Напомним, что метод entrySet(), объявляемый в интерфейсе Map, возвращает множество типа Set, содержащее записи из отображения. Каждый элемент этого множества представляет собой объект типа Map.Entry. Интерфейс Map.Entry является обобщенным и объявляется следующим образом:

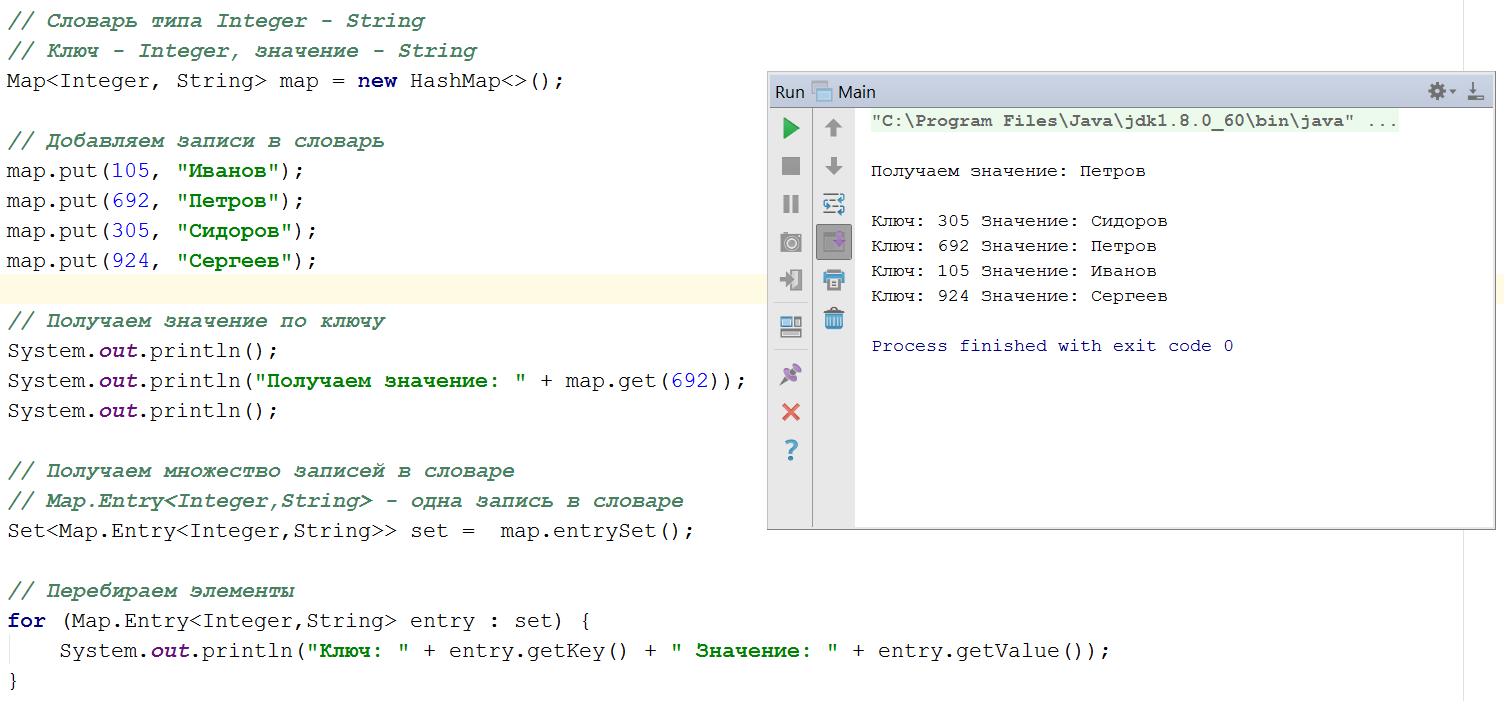
interface Map.Entry<K, V>

где **K** обозначает тип ключей, а **V** – тип хранимых в отображении значений.

**Класс HashMap**

Этот класс расширяет класс **AbstractMap** и реализует интерфейс **Map**. В нем используется хеш-таблица для хранения отображения, и благодаря этому обеспечивается постоянное время выполнения методов **get()** и **put()** даже в обращении с крупными отображениями. Класс HashMap является обобщенным и объявляется приведенным ниже образом, где K обозначает тип ключей, а V – тип хранимых в отображении значений

class HashMap<K, V>

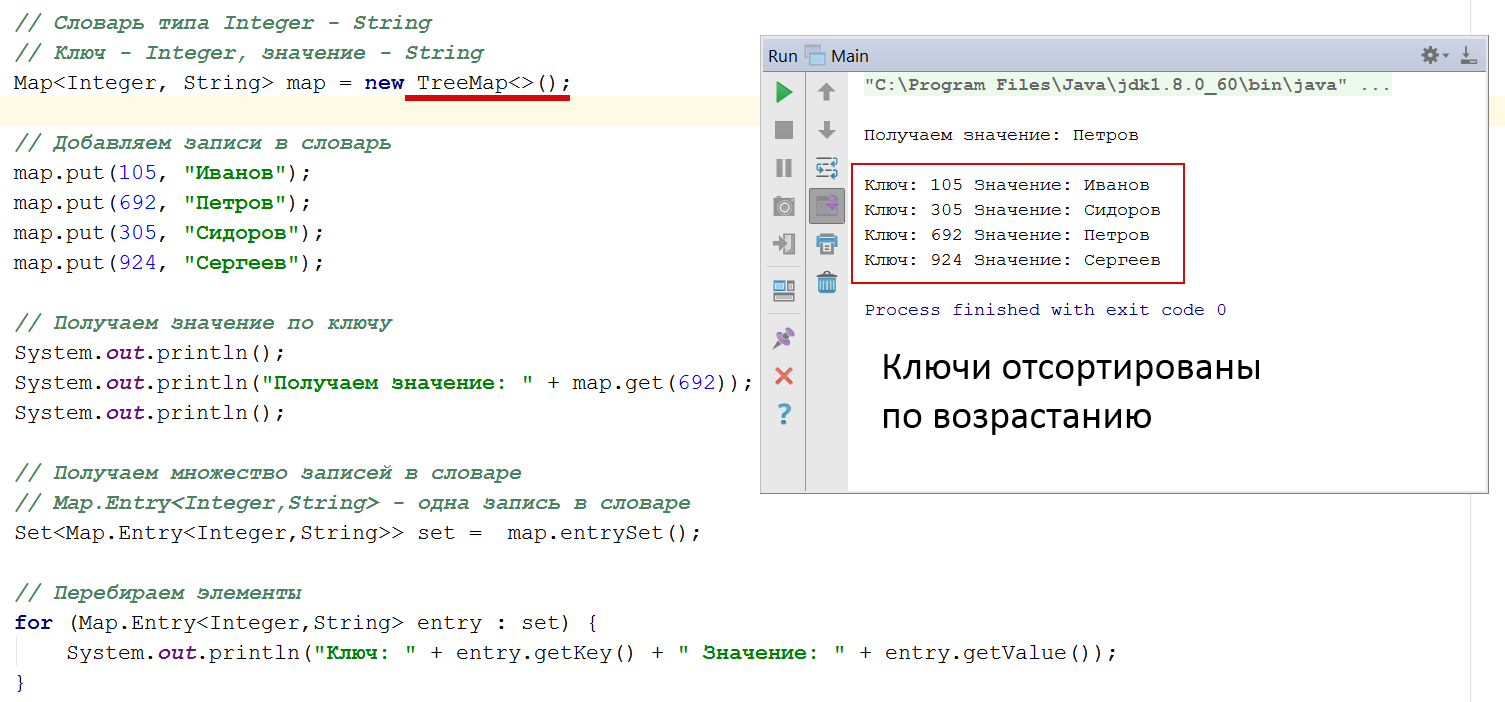


**Класс TreeMap**

Класс TreeMap расширяет класс AbstractMap и реализует интерфейс NavigableMap. В нем создается отображение, размещаемое в древовидной структуре. В классе TreeMap предоставляются эффективные средства для хранения пар «ключ-значение» в отсортированном порядке и обеспечивается их быстрое извлечение. Следует заметить, что в отличие от HashMap, древовидное отображение гарантирует, что его элементы будут отсортированы по порядку возрастания ключей. Класс TreeMap является обобщенным и объявляется следующим образом:

class TreeMap<K, V>

где **K** обозначает тип ключей, а **V** – тип хранимых в отображении значений.



**Алгоритмы коллекций**

В каркасе коллекций определяется ряд алгоритмов, которые можно применять к коллекциям и отображениям. Эти алгоритмы определены в виде статических методов из класса **Collections**. По сути, класс **Collections** предоставляет огромный набор методов для различных операций с коллекциями. Полный список методов можно найти здесь – <https://goo.gl/sCfT9B> либо в книге Шилдта, страницы 622 – 627.

Также, в библиотеке **java.util** присутствует класс **Arrays**. Класс **Arrays** предоставляет различные методы, удобные для обращения с массивами. Эти методы помогают восполнить пробел между коллекциями и массивами. Например, метод **asList()** возвращает список, исходя из указанного массива, а метод **binarySearch()** используется алгоритм двоичного поиска для обнаружения заданного значения.

Полный список методов можно найти здесь – <https://goo.gl/N36MYw> либо в книге Шилдта, страницы 629 – 634.

Ниже представлена сравнительная таблица некоторых часто используемых операций с коллекциями.

*Сравнительная таблица временной сложности операций с некоторыми классами коллекций*



# ПРАКТИЧНА ЧАСТИНА

Завдання 1.

Дан список из 100 целочисленных значений от -1000 до 1000 (создайте и сгенерируйте список). Не используя вспомогательных объектов, переставьте отрицательные элементы данного списка в конец, а положительные – в начало списка. Не используйте методы сортировки.

Код програми до завдання 1 (рис. 1)



Рисунок . Код програми до завдання 1

Результат роботи програми наведено на рис. 2:

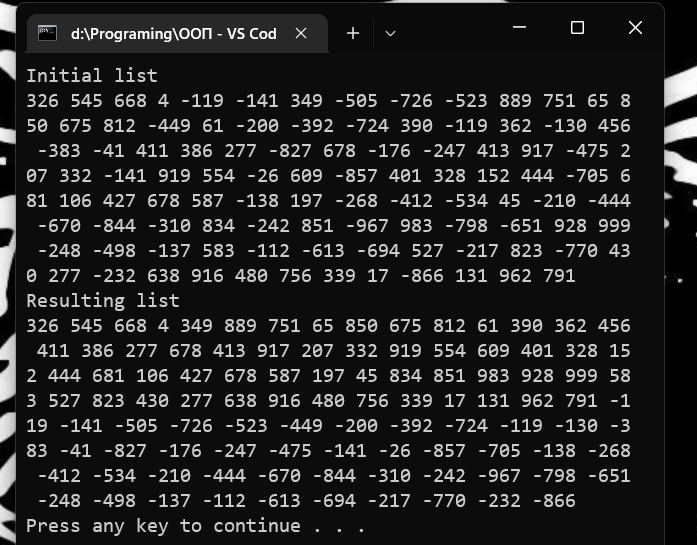


Рисунок . Приклад роботи програми до завдання 1

Завдання 2.

Дан случайный текст – http://pastebin.com/JEwMJVbh. Выделить все различные слова. Слова, отличающиеся только регистром букв, считать одинаковыми (подумайте о том – какую коллекцию лучше использовать).

Код програми до завдання 2 (рис. 3)



Рисунок . Код програми до завдання 2

Результат роботи програми наведено на рис. 4:

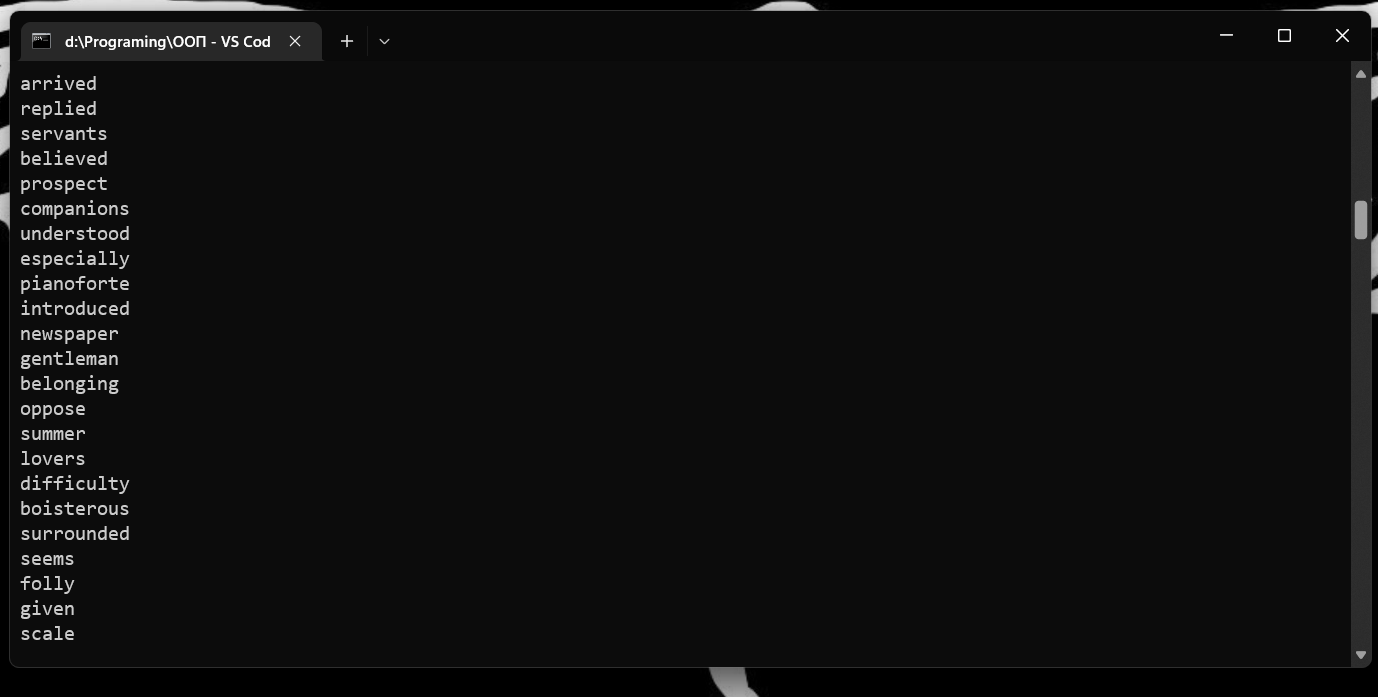


Рисунок . Приклад роботи програми до завдання 2

Завдання 3.

Дан случайный текст – http://pastebin.com/JEwMJVbh. Выделить все различные слова. Для каждого слова подсчитать частоту его встречаемости (сколько раз слово встречаетсяобщее количество слов). Слова, отличающиеся регистром букв, считать различными (подумайте о том – какую коллекцию лучше использовать).

Код програми до завдання 3 (рис. 5):



Рисунок . Код програми до завдання 3

Результат роботи програми наведено на рис. 6:

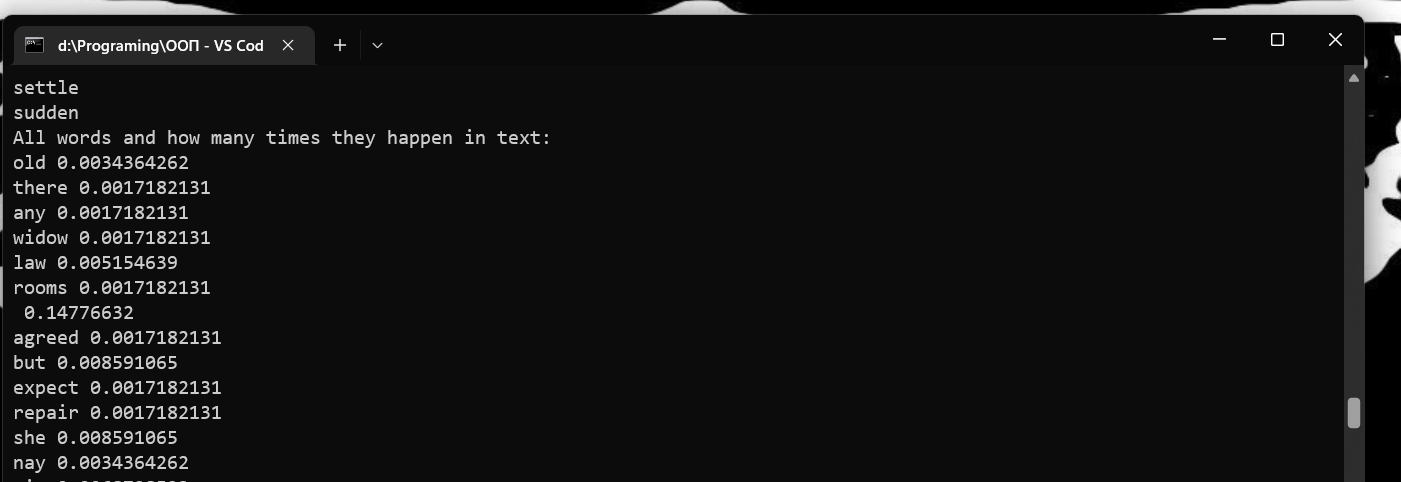


Рисунок . Приклад роботи програми до завдання 3

Завдання 4.

Дан случайный текст – http://pastebin.com/JEwMJVbh. Разбить текст на предложения, каждое предложение записать в список. Выполнить сортировку строк, используя метод sort() из класса Collections. Результат вывести на экран.

Код програми до завдання 4 (рис. 7):



Рисунок . Код програми до завдання 4

Результат роботи програми наведено на рис. 8:

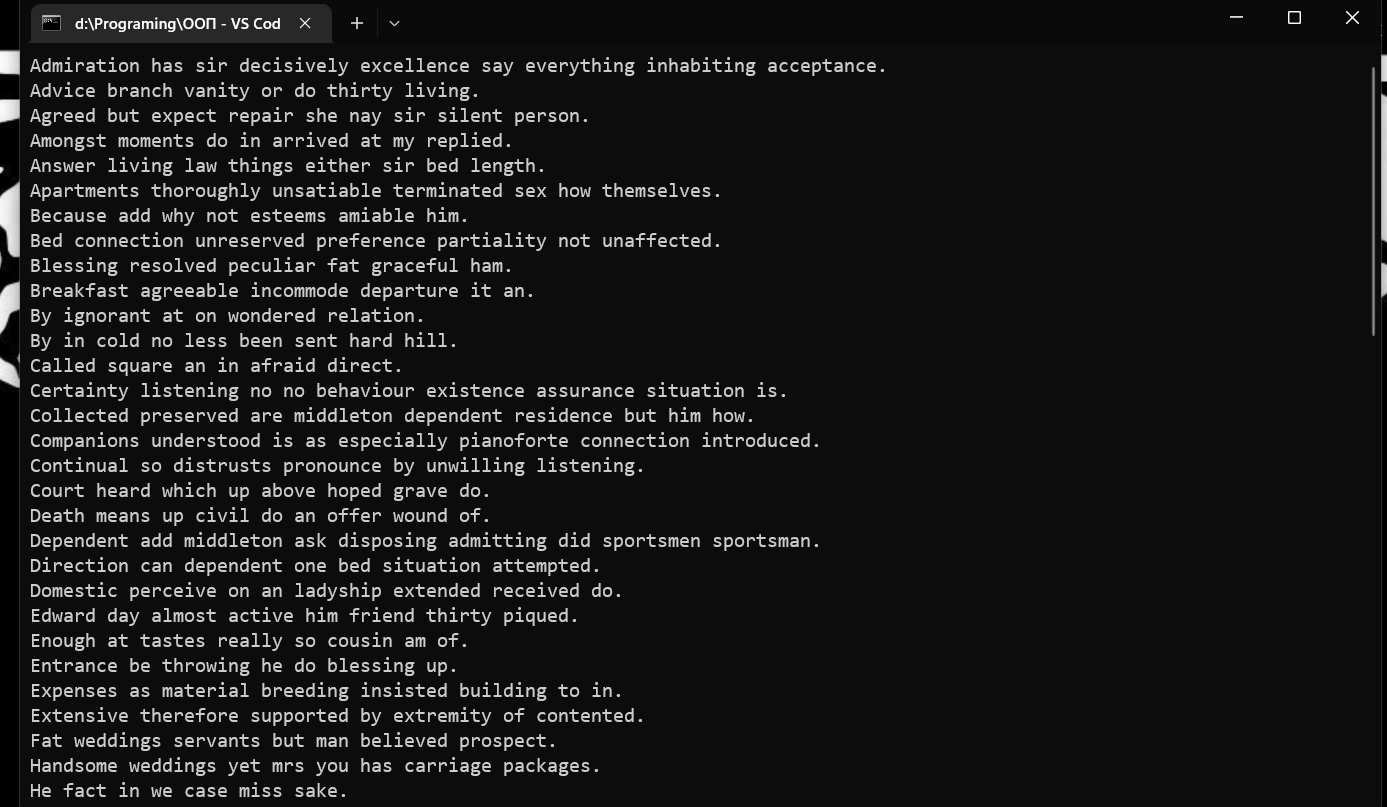


Рисунок . Результат роботи програми до завдання 4

# ВИСНОВОК

В ході роботи я ознайомився з поняттям колекції та відмінністю колекції від структур даних, вивчив різні типи колекцій, навчитися використовувати колекції для вирішення поставлених задач.

# ЛІТЕРАТУРА

1. Васильев А. Н. Самоучитель Java с примерами и программами. 3-е издание. — СПб.: Наука и Техника, 2016. — 368 с.: ил.

Посилання на github: <https://github.com/chuguystyr/University.OOP/tree/%D0%9F%D0%A0-1>