

**МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ  
КОММУНИКАЦИЙ**

**Ордена Трудового Красного Знамени**

**Федеральное государственное бюджетное образовательное учреждение  
высшего образования**

**«Московский технический университет связи и информатики»**

**Кафедра «Программная инженерия»**

**Отчет по лабораторной работе № 6.**

**по дисциплине «Информационные технологии и программирование»**

**по разделу:**

**«6. Работа с коллекциями»**

**Выполнила: студентка группы БПИ2401 Рябова Екатерина**

**Проверил: Харрасов Камиль Раисович**

**Москва**

**2025**

## Содержание

Цель работы: .....	2
Ход работы: .....	2
Задание 1 .....	2
Задание 2 .....	6
Задание 3 .....	9
Работа загружена на гитхаб по ссылке:.....	14
Ответы на контрольные вопросы: .....	14
Вывод: .....	24

### Цель работы:

Познакомиться с разными коллекциями в Java, научиться их использовать, а также научиться применять в работе с коллекциями дженерики.

### Ход работы:

#### Задание 1

*Написать программу, которая считывает текстовый файл и выводит на экран топ-10 самых часто встречающихся слов в это файле. Для решения задачи использовать коллекцию Map, где ключом будет слово, а значением — количество его повторений в файле.*

Реализуем класс TopWord, опираясь на предложенный в методичке пример. Используем HashMap. В нашем map ключ – String, значение – Integer. Считываем файл по словам, используя scanner.hasNext() для проверки конца файла и scanner.next() внутри цикла с условием для считывания. Так мы считаем файл по словам. Каждое слово мы приведем к нижнему регистру и удалим все ненужные символы (заменив их на пустую строку), при этом мы

оставим все апострофы – в начале, в середине и конце, а дефисы оставим только в середине. Используем для этого регулярное выражение: `^[^a-zA-Za-яА-Я']+|[^a-zA-Za-яА-Я']+$`, которые ищет символы кроме букв и апострофа в начале строки, которые встречаются 1 или более раз или символы кроме букв и апострофов в конце строки, которые встречаются 1 или более раз. Важно, что мы просто обрезаем всякого рода знаки препинания, а не обрабатываем ошибки ввода. То есть, например, в слове «слово» нолик удален не будет. Проверяем, что после удаления всех лишних символов в начале и конце у нас не пустая строка и обновляем в нашей Map по ключу, равному этому слову, значение, делая его на единицу больше чем было. Используем метод `getOrDefault(word, 0)`, который возвращает значение по ключу, если такой ключ есть, иначе 0. Закроем сканнер (после прочтения всех слов) и создадим список из элементов Map, то есть из `Entry <String, Integer>`. Используем метод `entrySet()`, который автоматически возвращает множество всех пар "ключ-значение" в виде объектов `Map.Entry`. Это необходимо, так как `HashMap` не итерируема. Для сортировки списка по убыванию количества повторений мы напишем анонимный класс, реализующий интерфейс `Comparator`, с одним единственным методом `compare`, который мы переопределим, так, чтобы он принимал на вход `entry`шки, и возвращал отрицательное число, если `o1 > o2` и положительное число, если `o1 < o2`, ( и 0, если они равны), то есть ровно обратный результат от работы функции `o1.getValue().compareTo(o2.getValue())`, то есть `o2.getValue().compareTo(o1.getValue())`. Далее выведем результат, используя счетчик (до 10) и проход по элементам списка.

*Листинг 1.1. Программа, которая считывает текстовый файл и выводит на экран топ-10 самых часто встречающихся слов в это файле.*

```
package sixthlab;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.*;
```

```

public class TopWord {
    public static void main(String[] args) {
        // указываем путь к файлу
        String filePath = "text.txt";

        // создаем объект File
        File file = new File(filePath);

        // создаем объект Scanner для чтения файла
        Scanner scanner = null;
        try {
            scanner = new Scanner(file);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            return;
        }

        // создаем объект Map для хранения слов и их количества
        Map<String, Integer> wordCountMap = new HashMap<>();

        // читаем файл по словам и добавляем их в Map
        while (scanner.hasNext()) {
            String word = scanner.next().toLowerCase().replaceAll("[^a-zA-Za-zA-я' ]+|^[^a-zA-Za-zA-яА-Я']+$", ""); // удаляем все символы, не являющиеся буквами и апострофами в начале и конце
            if (!word.isEmpty()) {
                wordCountMap.put(word, wordCountMap.getOrDefault(word, 0) + 1);
            }
        }

        // закрываем Scanner
        scanner.close();

        // создаем список из элементов Map
        List<Map.Entry<String, Integer>> list = new
        ArrayList<>(wordCountMap.entrySet());

        // сортируем список по убыванию количества повторений
        Collections.sort(list, new Comparator<Map.Entry<String, Integer>>() {
            @Override
            public int compare(Map.Entry<String, Integer> o1, Map.Entry<String,
            Integer> o2) {
                return o2.getValue().compareTo(o1.getValue()); // сортировка по
                убыванию
            }
        });

        System.out.println("Топ-10 самых часто встречающихся слов:");

        // выводим результат
    }
}

```

```

        int count = 0;
        for (Map.Entry<String, Integer> entry : list) {
            if (count < 10) {
                System.out.println((count + 1) + ". " + entry.getKey() + " - " +
entry.getValue() + " раз(a)");
                count++;
            } else {
                break;
            }
        }
    }
}

```

Результаты представлен на рисунках 1 и 2:

```

PS C:\Users\user\.vscode\mtuci\java\labs> java sixthlab.TopWord
Топ-10 самых часто встречающихся слов:
1. и - 173 раз(a)
2. в - 107 раз(a)
3. на - 52 раз(a)
4. не - 45 раз(a)
5. с - 41 раз(a)
6. хоббиты - 35 раз(a)
7. а - 27 раз(a)
8. но - 25 раз(a)
9. к - 23 раз(a)
10. они - 23 раз(a)

```

Рисунок 1. Результат корректного выполнения программы с ознакомительным фрагментом Властелина Колец.

```
Топ-10 самых часто встречающихся слов:  
1. test - 13 раз(a)  
2. don't - 12 раз(a)  
3. john_doe - 10 раз(a)  
4. word - 9 раз(a)  
5. dont - 8 раз(a)  
6. jane_doe - 8 раз(a)  
7. well-known - 7 раз(a)  
8. analysis - 7 раз(a)  
9. john_doe's - 7 раз(a)  
10. it's - 6 раз(a)
```

Рисунок 2. Результат корректного выполнения программы с текстом с разнообразными граничными случаями.

## Задание 2

*Написать обобщенный класс `Stack< T>` , который реализует стек на основе массива. Класс должен иметь методы `push` для добавления элемента в стек, `pop` для удаления элемента из стека и `peek` для получения верхнего элемента стека без его удаления.*

Реализуем класс `Stack` как показано в примере в методичке. У класса будет два поля – массив `T`-эшеч и размер, который будет указывать на индекс последнего элемента. В конструкторе мы будем создавать массив `Object` заданной длины и апкейстить его в массив `T`. В методах `push` и `pop` и `peek` мы будем проверять что массив не переполнен (или не пуст), затем двигать указатель на индекс последнего элемента в нужную сторону (кроме `peek`) и затем класть элемент в массив или доставать из него по индексу. Если мы достали элемент, то, чтобы удалить его просто положим на его место `null`. Допишем так же дополнительные методы – `isEmpty`, `size` и `isFull`, который будут возвращать, пуст ли массив, его размер и полон ли стек.

Листинг 2.1. Класс, который реализует стек на основе массива.

```

package stack;

public class Stack<T> {
    private T[] data;
    private int size;

    public Stack(int capacity) {
        data = (T[]) new Object[capacity];
        size = 0;
    }

    public void push(T element) {
        if (size >= data.length) {
            throw new StackOverflowError("Stack is full");
        }
        size++;
        data[size - 1] = element;
    }

    public T pop() {
        if (size == 0) {
            throw new IllegalStateException("Stack is empty");
        }

        T element = data[size - 1];
        data[size - 1] = null;
        size--;
        return element;
    }

    public T peek() {
        if (size == 0) {
            throw new IllegalStateException("Stack is empty");
        }
        return data[size - 1];
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public int size() {
        return size;
    }

    public boolean isFull() {
        return size == data.length;
    }
}

```

Напишем класс Main, проверяющий работу стека. Создадим стэки с разными типами данных, проверим работу всех методов.

*Листинг 2.2. Класс, который проверяет работу стэка.*

```
package stack;

public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>(10);
        stack.push(1);
        stack.push(2);
        stack.push(3);
        System.out.println(stack.pop());
        System.out.println(stack.peek());
        stack.push(4);
        System.out.println(stack.pop());

        Stack<String> stringStack = new Stack<>(5);
        stringStack.push("Hello");
        stringStack.push("World");
        System.out.println(stringStack.pop()); // World
        System.out.println(stringStack.peek()); // Hello

        Stack<Double> doubleStack = new Stack<>(3);
        System.out.println("Пуст ли стэк: " + doubleStack.isEmpty());
        System.out.println("Полон ли стэк: " + doubleStack.isFull());
        System.out.println("Размер стэка: " + doubleStack.size());
        try {
            doubleStack.pop();
        } catch (IllegalStateException e) {
            System.out.println("Ошибка: " + e.getMessage());
        }

        Stack<Character> charStack = new Stack<>(2);
        charStack.push('A');
        System.out.println("Пуст ли стэк: " + charStack.isEmpty());
        System.out.println("Полон ли стэк: " + charStack.isFull());
        System.out.println("Размер стэка: " + charStack.size());
        charStack.push('B');
        System.out.println("Пуст ли стэк: " + charStack.isEmpty());
        System.out.println("Полон ли стэк: " + charStack.isFull());
        System.out.println("Размер стэка: " + charStack.size());
        try {
            charStack.push('C');
        } catch (StackOverflowError e) {
            System.out.println("Ошибка: " + e.getMessage()); // Stack is full
        }
    }
}
```

```
}
```

Результат работы программы представлен на рисунке 3.

```
PS C:\Users\user\.vscode\mtuci\java\labs\sixthlab> java stack.Main
3
2
4
World
Hello
Пуст ли стэк: true
Полон ли стэк: false
Размер стэка: 0
Ошибка: Stack is empty
Пуст ли стэк: false
Полон ли стэк: false
Размер стэка: 1
Пуст ли стэк: false
Полон ли стэк: true
Размер стэка: 2
Ошибка: Stack is full
```

Рисунок 3. Результат корректной работы программы

### Задание 3

*Разработать программу для учета продаж в магазине. Программа должна позволять добавлять проданные товары в коллекцию, выводить список проданных товаров, а также считать общую сумму продаж и наиболее популярный товар.*

*Вариант 3: Использовать HashSet для хранения списка проданных товаров.*

Важно заметить, что HashSet не хранит дубликаты, а значит мы должны хранить вместе с названием товара информацию о проданных единицах. Но HashSet не хранит пары, как это делает Map (значением тут просто является заглушка) и возвращает элемент по этому ключу. Значит, будем хранить в HashSet экземпляры класса Product, у которых будут поля с ценой за единицу товара и кол-во проданных штук. Напишем конструктор для этого класса, который автоматически делает проданное кол-во равное 1, присваивает

позиции имя и цену. При добавлении продажи у нас будет увеличиваться счетчик проданных штук. Добавим геттеры для получения имени продукта, кол-ва проданных единиц, цены и общей стоимости для всех проданных товаров этого типа. Добавим сеттер для цены, переопределим методы сравнения и хэширования (для использования в HashSet). Так как в HashSet не могут лежать дубли, то сравнение и хэширование будем производить только по имени товара. Переопределим метод форматирования в строку информации о товаре.

Листинг 3.1. Класс Product.

```
package product;

import java.util.Objects;

public class Product {
    private String name;
    private int quantitySold;
    private double price;

    public Product(String name, double pricePerUnit) {
        this.name = name;
        this.price = pricePerUnit;
        this.quantitySold = 1;
    }

    public void addSale() {
        this.quantitySold++;
    }

    public String getName() { return name; }
    public int getQuantitySold() { return quantitySold; }
    public double getPrice() { return price; }
    public double getTotalRevenue() { return quantitySold * price; }

    public void setPrice(double newPrice) {
        this.price = newPrice;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || !(o instanceof Product)) return false;
        Product product = (Product) o;
        return product.name.equals(name);
    }
}
```

```

    }

    @Override
    public int hashCode() {
        return Objects.hash(name);
    }

    @Override
    public String toString() {
        return String.format("%s: %d шт. * %.2f руб. = %.2f руб.",
                               name, quantitySold, price, getTotalRevenue());
    }
}

```

Напишем основной класс – Store с полями сета из продуктов и общей стоимости. В конструкторе будем создавать пустой HashSet и приравнивать общую сумму к 0. Напишем метод поиска продукта. В этом методе мы будем итерировать наш HashSet и сравнивать переданное в качестве параметров название с названиями продуктов, лежащих внутри данной коллекции. Добавим метод продажи продукта. В нем будем проверять существование данного продукта (с помощью поиска), если такой позиции нет – будем создавать новый продукт, если есть, то метод поиска вернет нам этот продукт, и мы просто воспользуемся методом добавления продажи класса Product (При этом, мы обновим цену, если она не совпадет с уже указанной ценой). В обоих случаях общая цена будет увеличена на данную. Добавим функцию выведения списка проданных товаров – просто итерация по коллекции с использованием метода toString для producta. Добавим геттер для общей стоимости. Добавим метод для определения наиболее популярного товара. В этом методе мы будем проходить по HashSet и сравнивать наибольшее найденное кол-во проданных товаров с этим значением на каждой позиции. Вернем строковое представление товара с наибольшим количеством продаж.

*Листинг 3.2. Класс Store..*

```
package product;
```

```

import java.util.HashSet;
import java.util.Set;

public class Store {
    private Set<Product> products;
    private double totalRevenue;

    public Store() {
        products = new HashSet<>();
        totalRevenue = 0.0;
    }

    public void addSale(String productName, double price) {
        Product existing = findProduct(productName);
        if (existing != null) {
            existing.addSale();
            existing.setPrice(price);
        } else {
            Product newProduct = new Product(productName, price);
            products.add(newProduct);
            System.out.println("Добавлен новый товар: " + productName);
        }

        totalRevenue += price;
    }

    private Product findProduct(String productName) {
        for (Product product : products) {
            if (product.getName().equals(productName)) {
                return product;
            }
        }
        return null;
    }

    public void SoldProducts() {
        if (products.isEmpty()) {
            System.out.println("Проданных товаров нет");
            return;
        }

        int counter = 1;
        for (Product product : products) {
            System.out.println(Integer.toString(counter) + ". " +
product.toString());
            counter++;
        }
    }

    public double getTotalSales() {
        return totalRevenue;
    }

    public String getMostPopularProduct() {
        if (products.isEmpty()) {

```

```

        return "Нет данных о продажах";
    }

    Product mostPopular = null;
    int maxQuantity = 0;

    for (Product product : products) {
        if (product.getQuantitySold() > maxQuantity) {
            maxQuantity = product.getQuantitySold();
            mostPopular = product;
        }
    }

    return mostPopular.toString();
}
}

```

Напишем класс Main для тестирования работы этих классов. Создадим новый магазин и насоздаем продаж для разных продуктов, и создавая новые позиции, и просто добавляя кол-во проданных штук. Выведем информацию о проданных товарах, общую цену и самый популярный товар.

Листинг 3.3. Класс Main.

```

package product;

public class Main {
    public static void main(String[] args) {
        Store store = new Store();

        store.addSale("Хлеб", 50.0);
        store.addSale("Молоко", 80.0);
        store.addSale("Хлеб", 50.0);
        store.addSale("Хлеб", 50.0);
        store.addSale("Яйца", 120.0);
        store.addSale("Шоколад", 60.0);
        store.addSale("Шоколад", 60.0);
        store.addSale("Шоколад", 60.0);
        store.addSale("Шоколад", 60.0);
        store.addSale("Хлеб", 50.0);
        store.addSale("Молоко", 80.0);
        store.addSale("Хлеб", 50.0);
        store.addSale("Яйца", 80.0);
    }
}

```

```

        store.SoldProducts();
        System.out.println(store.getTotalSales());
        System.out.println(store.getMostPopularProduct());
    }
}

```

Результат работы программы представлен на рисунке 4:

```

PS C:\Users\user\.vscode\mtuci\java\labs\sixthlab> java product.
Main
Добавлен новый товар: Хлеб
Добавлен новый товар: Молоко
Добавлен новый товар: Яйца
Добавлен новый товар: Шоколад
1. Хлеб: 5 шт. * 50,00 руб. = 250,00 руб.
2. Яйца: 2 шт. * 80,00 руб. = 160,00 руб.
3. Молоко: 2 шт. * 80,00 руб. = 160,00 руб.
4. Шоколад: 4 шт. * 60,00 руб. = 240,00 руб.
850.0
Хлеб: 5 шт. * 50,00 руб. = 250,00 руб.

```

Рисунок 4. Результат корректной работы программы.

Можно заметить, что общая цена не совпадает по итоговой цене по каждой позиции. Это связано с тем, что общая цена обновляется в момент продажи, т.е. учитывает цену на момент покупки, а итоговая сумма по каждой позиции рассчитывается исходя из последней установленной цены на товар.

Работа загружена на гитхаб по ссылке:  
[https://github.com/Kateriabova/Riaboava\\_Kate\\_BPI2401\\_IT2.git](https://github.com/Kateriabova/Riaboava_Kate_BPI2401_IT2.git)

## Ответы на контрольные вопросы:

### 1. Какие интерфейсы коллекций есть в Java?

В Java Framework существует несколько основных интерфейсов:

- Collection - корневой интерфейс для большинства коллекций. Определяет базовые операции: добавление, удаление, проверка размера и т.д.
- List - интерфейс для упорядоченных коллекций. Элементы имеют индекс, допускаются дубликаты. Примеры: ArrayList, LinkedList.
- Set - интерфейс для коллекций уникальных элементов (без дубликатов). Не гарантирует порядок элементов. Примеры: HashSet, TreeSet.
- Queue - интерфейс для реализации очереди (FIFO - First In First Out). Примеры: PriorityQueue, LinkedList.
- Deque (Double Ended Queue) - двусторонняя очередь, позволяет добавлять/удалять элементы с обоих концов.
- Map - интерфейс для хранения пар "ключ-значение". Не наследует интерфейс Collection. Примеры: HashMap, TreeMap.
- SortedSet и SortedMap - расширения Set и Map соответственно, которые гарантируют отсортированный порядок элементов.).

## 2. Какие классы коллекций есть в Java?

Для List

- ArrayList - список на основе массива, быстрый доступ по индексу
- LinkedList - двусвязный список, быстрая вставка/удаление
- Vector - синхронизированный аналог ArrayList (устарел)
- Stack - стек на основе Vector

Для Set

- HashSet - множество на основе хэш-таблицы, не гарантирует порядок
- LinkedHashSet - HashSet, сохраняющий порядок добавления
- TreeSet - множество с сортировкой элементов (красно-черное дерево)

Для Map:

- HashMap - хэш-таблица, не гарантирует порядок
- LinkedHashMap - HashMap, сохраняющий порядок добавления
- TreeMap - отсортированная по ключам карта (красно-черное дерево)
- Hashtable - синхронизированный аналог HashMap

Для Queue/Deque:

- PriorityQueue - очередь с приоритетами (сортировка)
- ArrayDeque - двусторонняя очередь на основе массива

### **3. Что такое итератор?**

Итератор (Iterator) - это объект, который позволяет последовательно обходить элементы коллекции без знания её внутренней структуры.

Основные методы:

- hasNext() - проверяет, есть ли следующий элемент
- next() - возвращает следующий элемент
- remove() - удаляет текущий элемент (опциональный).

### **4. Как работают коллекции на основе интерфейса Map?**

Map хранит данные в виде пар "ключ-значение". Основные характеристики:

- Ключи уникальны (дубликаты не допускаются)
- Один ключ соответствует одному значению
- Быстрый поиск по ключу (обычно  $O(1)$  для HashMap)
- Основные реализации: HashMap, TreeMap, LinkedHashMap

### **5. Как работают коллекции на основе интерфейса List?**

List - это упорядоченная коллекция с индексами:

- Элементы имеют порядок (индексы от 0 до size-1)

- Допускаются дубликаты
- Доступ по индексу за  $O(1)$  в ArrayList,  $O(n)$  в LinkedList
- Основные операции: добавление, удаление, получение по индексу.

## 6. Как работают коллекции на основе интерфейса Set?

Set - коллекция уникальных элементов:

- Не допускает дубликаты (проверка через equals())
- Не гарантирует порядок элементов (кроме LinkedHashSet и TreeSet)
- Быстрая проверка наличия элемента (обычно  $O(1)$ )
- Используется, когда важна уникальность

## 7. Как можно синхронизировать коллекции в Java?

Способы синхронизации:

1. Использовать синхронизированные версии (устарело):

```
List<String> syncList = Collections.synchronizedList(new ArrayList<>());
Map<String, String> syncMap = Collections.synchronizedMap(new
HashMap<>());
```

2. Использовать Concurrent коллекции:

```
ConcurrentHashMap<String, Integer> map = new
ConcurrentHashMap<>();
CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>()
```

3. Использовать блоки synchronized:

```
synchronized(list) {
    list.add(element);
}
```

4. Использовать явные блокировки (ReentrantLock):

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    // операции с коллекцией
} finally {
    lock.unlock();
}
```

## 8. Какие методы предоставляет интерфейс Collection?

Основные методы интерфейса Collection:

- Добавление: `add(E e)`, `addAll(Collection<? extends E> c)`
- Удаление: `remove(Object o)`, `removeAll(Collection<?> c)`, `clear()`
- Проверка: `contains(Object o)`, `containsAll(Collection<?> c)`, `isEmpty()`
- Размер: `size()`
- Преобразование: `toArray()`, `toArray(T[] a)`
- Итерация: `iterator()`, `forEach(Consumer<? super E> action)`
- Потoki: `stream()`, `parallelStream()`
- Сохраняющие операции: `retainAll(Collection<?> c)`

## 9. Какие реализации интерфейса List вы знаете?

Основные реализации List:

- `ArrayList` - динамический массив
  - Плюсы: быстрый доступ по индексу  $O(1)$
  - Минусы: медленная вставка/удаление в середине  $O(n)$
- `LinkedList` - двусвязный список
  - Плюсы: быстрая вставка/удаление  $O(1)$
  - Минусы: медленный доступ по индексу  $O(n)$
- `Vector` (устарел) - синхронизированный `ArrayList`
- `Stack` (устарел) - стек на основе `Vector`
- `CopyOnWriteArrayList` - потокобезопасный список для редких модификаций

## 10. Какие реализации интерфейса Set вы знаете?

Основные реализации Set:

- `HashSet` - на основе `HashMap`
  - Не гарантирует порядок
  - Операции  $O(1)$  в среднем случае

- `LinkedHashSet` - `HashSet` + сохранение порядка добавления
  - Гарантирует порядок вставки
  - Немного медленнее `HashSet`
- `TreeSet` - на основе `TreeMap` (красно-черное дерево)
  - Элементы отсортированы
  - Операции  $O(\log n)$
- `EnumSet` - для enum типов, очень эффективный
- `CopyOnWriteArraySet` - потокобезопасный Set

## 11. Что такое `Comparable` и `Comparator`?

`Comparable` - интерфейс, который реализует сам класс для определения "естественного" порядка сортировки:

```
class Person implements Comparable<Person> {
    private String name;
    @Override
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
}
```

`Comparator` - отдельный класс для сравнения, позволяет задавать разные стратегии сортировки:

```
Comparator<Person> byAge = new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        return Integer.compare(p1.getAge(), p2.getAge());
    }
};
```

```
Collections.sort(people, byAge);
```

## 12. Что такое параметр типа?

Параметр типа (type parameter) - это placeholder для конкретного типа в обобщенных классах, интерфейсах и методах. Обозначается в угловых скобках:

```

public class Box<T> { // T - параметр типа
    private T value;

    public void set(T value) {
        this.value = value;
    }

    public T get() {
        return value;
    }
}

```

```

Box<Integer> intBox = new Box<>(); // Integer - аргумент типа

```

### 13. Как параметризовать метод, класс?

Параметризация класса:

```

public class Pair<K, V> { // Два параметра типа
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
}

```

```

Pair<String, Integer> pair = new Pair<>("age", 25);"

```

Параметризация метода:

```

public class Utils {
    // Параметр типа объявляется перед возвращаемым типом
    public static <T> T getFirst(List<T> list) {
        if (list.isEmpty()) return null;
        return list.get(0);
    }

    // Метод с несколькими параметрами типа
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
            p1.getValue().equals(p2.getValue());
    }
}

```

```
}  
}
```

#### 14. Что такое стирание типов?

Стирание типов (type erasure) - процесс, при котором компилятор Java удаляет информацию о параметрах типов после компиляции:

```
// Исходный код (compile-time)  
List<String> strings = new ArrayList<>();  
String s = strings.get(0);  
  
// После стирания (runtime)  
List strings = new ArrayList(); // Raw type  
String s = (String) strings.get(0); // Приведение типа
```

#### 15. Как можно обойти ограничения стирания типов?

Передача Class объекта:

```
public <T> T createInstance(Class<T> clazz) throws Exception {  
    return clazz.newInstance();  
}
```

Использование фабрик:

```
public interface Factory<T> {  
    T create();  
}  
  
public <T> T create(Factory<T> factory) {  
    return factory.create();  
}
```

Массивы через рефлексия:

```
public <T> T[] createArray(Class<T> clazz, int size) {  
    return (T[]) Array.newInstance(clazz, size);  
}
```

Хранение информации о типе через суперкласс:

```
public abstract class TypeReference<T> {  
    private final Type type;
```

```

        protected TypeReference() {
            Type superclass = getClass().getGenericSuperclass();
            this.type = ((ParameterizedType)
superclass).getActualTypeArguments()[0];
        }
    }

```

## 16. Как работают дженерики с массивами?

Нельзя создать массив параметризованного типа:

```

// Не компилируется!
List<String>[] array = new List<String>[10];

```

Можно создать массив raw type и привести:

```

List<String>[] array = (List<String>[]) new List[10]; // Предупреждение

```

Можно использовать массивы с wildcard:

```

List<?>[] array = new List<?>[10]; // Допустимо

```

Лучше использовать коллекции вместо массивов:

```

List<List<String>> list = new ArrayList<>();

```

## 17. Можно ли создать массив дженериков?

Прямо нельзя, но есть обходные пути:

```

// 1. Создание массива raw type с приведением
T[] array = (T[]) new Object[size];

```

```

// 2. Использование рефлексии (нужен Class<T>)
public static <T> T[] createGenericArray(Class<T> clazz, int size) {
    return (T[]) Array.newInstance(clazz, size);
}

```

```

// 3. Хранение Object[] и приведение при извлечении
public class GenericArray<T> {
    private Object[] array;

    public GenericArray(int size) {
        array = new Object[size];
    }
}

```

```

    }

    @SuppressWarnings("unchecked")
    public T get(int index) {
        return (T) array[index];
    }

    public void set(int index, T value) {
        array[index] = value;
    }
}

```

## 18. Что такое wildcard тип?

Wildcard (подстановочный тип) - обозначается ? и представляет неизвестный тип:

```

// 1. Неограниченный wildcard - любой тип
List<?> list = new ArrayList<String>();

```

```

// 2. Upper bounded wildcard - тип или его подтипы
List<? extends Number> numbers = new ArrayList<Integer>(); // Integer
extends Number

```

```

// 3. Lower bounded wildcard - тип или его супертипы
List<? super Integer> integers = new ArrayList<Number>(); // Number
super Integer]

```

- List<?> можно читать как Object, но нельзя добавлять (кроме null)
- List<? extends T> можно читать как T, но нельзя добавлять
- List<? super T> можно добавлять T, но читать только как Object

## 19. В чем разница между <? extends T> и <? super T>??

PECS принцип (Producer Extends, Consumer Super):

? extends T (Producer):

- Коллекция "производит" элементы типа T
- Можно читать элементы как T
- Нельзя добавлять элементы (кроме null)

- Используется, когда коллекция является источником данных

```
// Метод только читает из коллекции
public static double sum(List<? extends Number> numbers) {
    double sum = 0;
    for (Number n : numbers) {
        sum += n.doubleValue();
    }
    return sum;
}
```

? super T (Consumer):

- Коллекция "потребляет" элементы типа T
- Можно добавлять элементы типа T
- Читать можно только как Object
- Используется, когда коллекция является приемником данных

```
// Метод только записывает в коллекцию
public static void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i); // Можно добавить Integer
    }
}
```

## **Вывод:**

В ходе работы мы изучили коллекции в Java и их применения, научились использовать дженерики.