МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ордена трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский технический университет связи и информатики»

Кафедра Математическая кибернетика и информационные технологии

Отчет по лабораторной работе № 3

Выполнил: студент группы БПИ2401
Трухина Анастасия Александровна
Проверил: Харрасов Камиль Раисович

Москва,
2025

Оглавление

Цель работы:
Изучить базовый класс Object в Java, принципы работы его методов (в частности, equals и hashCode), а также структуру и применение хэш-таблиц, включая реализацию собственной хэш-таблицы и использование встроенных коллекций типа HashMap.

Задание:
Задание 1.

1. Создайте класс HashTable, который будет реализовывать хэш-таблицу с помощью метода цепочек.

2. Реализуйте методы put(key, value), get(key) и remove(key), которые добавляют, получают и удаляют пары «ключ-значение» соответственно.

3. Добавьте методы size() и isEmpty(), которые возвращают количество элементов в таблице и проверяют, пуста ли она.

Задание 2.

Реализация хэш-таблицы для учета заказов в интернет магазине. Ключом будет номер заказа, а значением — объект класса Order, содержащий информацию о товарах, адресе доставки и стоимости заказа. Необходимо реализовать операции вставки, поиска и удаления заказа по номеру заказа

Основная часть

Задание 1:

Хэш-таблица:

```
package Laba3;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

class HashTable<K, V> {
    private List<LinkedList<Entry<K, V>>> table;
    private int size;

    public HashTable() {
        table = new ArrayList<>();
        for (int i = 0; i < 16; i++) {
            table.add(null);
        }
        size = 0;
    }

    private int hash(K key) {
```

```
    return Math.abs(key.hashCode()) % table.size();
}

public void put(K key, V value) {
    int index = hash(key);

    if (table.get(index) == null) {
        table.set(index, new LinkedList<Entry<K, V>>());
    }

    for (Entry<K, V> entry : table.get(index)) {
        if (entry.key.equals(key)) {
            entry.value = value;
            return;
        }
    }

    table.get(index).add(new Entry<>(key, value));
    size++;
}

public V get(K key) {
    int index = hash(key);
    LinkedList<Entry<K, V>> bucket = table.get(index);

    if (bucket != null) {
        for (Entry<K, V> entry : bucket) {
            if (entry.key.equals(key)) {
                return entry.value;
            }
        }
    }
    return null;
}

public V remove(K key) {
    int index = hash(key);
    LinkedList<Entry<K, V>> bucket = table.get(index);

    if (bucket != null) {
        for (int i = 0; i < bucket.size(); i++) {
            Entry<K, V> entry = bucket.get(i);
            if (entry.key.equals(key)) {
                V removedValue = entry.value;
                bucket.remove(i);
                size--;
                return removedValue;
            }
        }
    }
    return null;
}
```

```
    public int size() {
        return size;
    }

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

Entry класс с generics:

```
package Laba3;

class Entry<K, V> {
    K key;
    V value;

    Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
}
```

Задание 2

Item класс

```
package Laba3;

class Item {
    String name;
    double price;

    Item(String name, double price) {
        this.name = name;
        this.price = price;
    }

    @Override
    public String toString() {
        return name + " - " + price;
    }
}
```

Order класс

```
package Laba3;

import java.util.ArrayList;
import java.util.List;

class Order {
    String orderNumber;
```

```
    List<Item> items;
    String deliveryAddress;
    double totalCost;
    String status;

    Order(String orderNumber, String deliveryAddress) {
        this.orderNumber = orderNumber;
        this.deliveryAddress = deliveryAddress;
        this.items = new ArrayList<>();
        this.totalCost = 0.0;
        this.status = "Новый";
    }

    void addItem(Item item) {
        items.add(item);
        totalCost += item.price;
    }

    void updateStatus(String status) {
        this.status = status;
    }

    @Override
    public String toString() {
        return "Order{" + orderNumber + ", " + items + ", " + deliveryAddress + ", " + totalCost +
", " + status + "}";
    }
}
```

Файл Main

```
package Laba3;

import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        HashTable<String, Integer> myTable = new HashTable<>();

        myTable.put("one", 1);
        myTable.put("two", 2);
        myTable.put("three", 3);

        System.out.println("Size: " + myTable.size());
        System.out.println("Get 'two': " + myTable.get("two"));
        System.out.println("Removed 'two': " + myTable.remove("two"));
        System.out.println("Size after remove: " + myTable.size());
        System.out.println("Is empty: " + myTable.isEmpty());
```

```
        HashTable<String, Order> orders = new HashTable<>();

        Order order1 = new Order("ORD001", "Москва");
        order1.addItem(new Item("Книга", 500.0));

        Order order2 = new Order("ORD002", "СПб");
        order2.addItem(new Item("Мышь", 300.0));

        orders.put(order1.orderNumber, order1);
        orders.put(order2.orderNumber, order2);

        System.out.println("Найден заказ: " + orders.get("ORD001"));

        order1.updateStatus("Отправлен");
        System.out.println("После изменения статуса: " + orders.get("ORD001"));

        orders.remove("ORD002");
        System.out.println("Размер после удаления: " + orders.size());
    }
}
```

Ответы на контрольные вопросы:

1. Класс Object является корнем всей иерархии классов в языке Java. Это означает, что каждый класс, включая пользовательские, автоматически наследует класс Object, даже если явного указания на это нет. Благодаря этому, каждый объект в Java обладает набором базовых методов, определенных в классе Object. Это включает такие методы, как equals(), hashCode(), toString(), getClass(), clone(), а также методы для синхронизации wait(), notify(), notifyAll(). Наследование от Object обеспечивает единообразие и предоставляет универсальные возможности для всех объектов в системе.

2. Метод equals() по умолчанию сравнивает ссылки на объекты, то есть проверяет, являются ли две переменные ссылками на один и тот же объект в памяти. Это поведение часто недостаточно, когда нужно сравнить содержимое объектов. Например, два разных объекта Person с одинаковыми полями name и age логично считать равными. Для этого equals() нужно переопределить. Метод hashCode() переопределяют, потому что хэш-коллекции (HashMap, HashSet) используют хеш-коды для эффективного хранения и поиска элементов. Согласно контракту, если два объекта равны по equals(), они обязаны возвращать одинаковый хеш-код. Несоблюдение этого правила приведет к неправильной работе коллекций, например, объект может быть добавлен, но не найден позже.

3. При переопределении equals() нужно соблюдать несколько важных правил, известных как контракт equals(). Метод должен быть рефлексивным: x.equals(x) всегда должно

возвращать true. Он должен быть симметричным: если x.equals(y) возвращает true, то y.equals(x) тоже должен возвращать true. Также требуется транзитивность: если x.equals(y) и y.equals(z) возвращают true, то и x.equals(z) должно возвращать true. Метод должен быть согласованным: при отсутствии изменений в объектах, результат вызова equals() должен оставаться постоянным. Наконец, для любого ненулевого объекта x, вызов x.equals(null) должен возвращать false. При переопределении hashCode() необходимо строго следовать контракту: равные объекты (согласно equals()) обязаны возвращать одинаковый хеш-код. Неравные объекты могут возвращать одинаковый хеш-код (коллизия), но для эффективности лучше, чтобы разные объекты возвращали разные хеш-коды.

4. Метод toString() возвращает строковое представление объекта. По умолчанию он возвращает имя класса и шестнадцатеричное представление хеш-кода объекта, например, Person@15db9742. Такое строковое представление не несёт полезной информации о состоянии объекта. Его часто переопределяют, чтобы получить человеко-читаемую и информативную строку, описывающую текущие значения полей объекта. Это чрезвычайно удобно для отладки кода, логирования действий программы и вывода информации о состоянии объекта в консоль или в интерфейсе.

5. Метод finalize() вызывался сборщиком мусора перед тем, как объект физически удалялся из памяти. Его можно было использовать для выполнения очистки, например, закрытия файловых дескрипторов или сетевых соединений, если объект владел ими. Однако использование finalize() считается устаревшим, потому что поведение и время его вызова непредсказуемы. Это может привести к утечкам ресурсов, проблемам с производительностью и непредсказуемому поведению программы. Современные альтернативы, такие как try-with-resources и API Cleaner, предоставляют более надежные и предсказуемые способы управления ресурсами.

6. Коллизия — это ситуация, которая возникает в хэш-таблице, когда два или более разных ключа при вычислении хеш-функции получают один и тот же индекс в массиве хранения. Это неизбежный побочный эффект работы хэш-функций, так как количество возможных ключей обычно гораздо больше, чем количество возможных индексов в массиве. Коллизии не означают ошибку, но требуют специального способа разрешения, чтобы оба (или более) значения могли быть корректно сохранены и найдены.

7. Существует несколько способов разрешения коллизий. Один из наиболее распространённых — метод цепочек (chaining). В этом методе каждая ячейка массива хэш-таблицы (корзина) содержит структуру данных, например, связный список, для хранения всех пар "ключ-значение", которые попадают в эту ячейку. При коллизии новая пара добавляется в список. Другой способ — открытая адресация (open addressing). При коллизии в открытой адресации алгоритм ищет другую свободную ячейку в массиве по

определённой стратегии (например, линейный пробинг, квадратичный пробинг, двойное хеширование) и помещает пару туда.

8. Хэш-таблица — это структура данных, предназначенная для хранения пар "ключ-значение". Внутри она использует массив "корзин" (buckets). Ключ передаётся в хеш-функцию, которая вычисляет индекс массива, где должна храниться пара. Значение сохраняется по этому индексу. Если два разных ключа получают одинаковый индекс (коллизия), используется один из методов разрешения коллизий, например, цепочки, где в одной корзине хранится список пар. Это позволяет эффективно находить, добавлять и удалять элементы по ключу.

9. Если в хэш-таблицу (например, в HashMap) добавить элемент с ключом, который уже существует в таблице, старое значение, связанное с этим ключом, будет заменено новым значением. Это стандартное поведение метода put(). Сравнение ключей при этом производится с помощью метода equals() объекта ключа. Таким образом, таблица всегда содержит только одно значение для каждого уникального ключа (с точки зрения equals()).

10. Если в хэш-таблицу добавить элемент с ключом, который имеет такой же хеш-код, что и у другого ключа, но при этом фактически это разные ключи (сравнение по equals() возвращает false), это также приведёт к коллизии. Эта новая пара "ключ-значение" будет добавлена в ту же "корзину", что и пара с одинаковым хеш-кодом (например, в список, если используется метод цепочек). При поиске значение будет найдено правильно, потому что после определения корзины по хеш-коду, происходит перебор элементов в корзине, и ключи сравниваются с помощью метода equals(), а не только по хеш-коду.

11. HashMap изменяется при достижении порогового значения, которое вычисляется как capacity loadFactor. LoadFactor (коэффициент загрузки) — это мера того, насколько заполнена таблица, прежде чем произойдёт увеличение размера. Когда количество элементов (size) превышает порог, происходит процесс, называемый resize (изменение размера). Внутри HashMap увеличивает свою внутреннюю ёмкость (capacity), обычно в 2 раза. Затем все существующие элементы перехешируются и помещаются в новую, большую таблицу. Это делается для поддержания эффективности операций (чтобы средняя длина цепочек или количество коллизий оставалось низким).

Заключение
Вывод:

В ходе лабораторной работы изучены особенности базового класса Object и реализована собственная хэш-таблица с методом цепочек. Рассмотрено переопределение методов equals и hashCode. Выполнена реализация системы учёта заказов с использованием

разработанной структуры данных.

Ссылка на ГитХаб с файлами кода: NT-005-TN/ITiP_LabWorks_Trukhina