Лабораторная работа № 3
Тема: «Основы работы с классами и объектами. Инкапсуляция. Разработка структур данных с помощью классов»

ЦЕЛЬ ЛАБОРАТОРНОЙ РАБОТЫ:

- Ознакомиться с основами написания классов;
- Освоить инкапсуляцию классов;
- Разобраться с созданием классов и методов на примере реализации различных структур данных.

ХОД РАБОТЫ:

Код с подробными комментариями можно найти по адресу
https://github.com/NickGodov/OOPLABS
можете сделать git clone либо скачать архив, нажмите кнопку «Download ZIP» внизу справа.

Сначала приведем пример описания класса

```
package ua.opu;

/**
 * Это "импорты", подключение сторонних классов для нашей программы.
 * Это примерный аналог include в языке C.
 * В данном случае, мы подключаем класс Time и LocalTime.
 * После слова import идет название пакета.
 * Если кто-то захочет подключить наш класс, он должен будет написать
 * import ua.opu.TimeInterval;
 *
 * Импорты идут сразу после package
 */
import java.sql.Time;
import java.time.LocalTime;

/**
 * Каждый класс (кроме внутренних) должен быть описан в своем отдельном файле
 * с расширением .java, имя файла должно совпадать с именем класса (с учетом регистра).
 * Класс должен начинаться с большой буквы, обычно это имя существительное.
 * Если имя класса состоит из нескольких слов, каждое слово начинается с большой буквы
 * Пример правильно названных классов: Stack, ConnectionManager, BinaryTree
 * Пример НЕПРАВИЛЬНО названных классов: logging, MYCLASS, Big Dog, run
 *
 * Синтаксис объявления класса: [модификатор доступа] class [ИмяКласса] {
 * тело класса
 * }
 */
public class TimeInterval {
```

```
    /** Первыми идут поля класса (переменные) */

    /**
     * Члена класса могут иметь модификаторы доступа
     * Модификаторы указывают область "видимости" переменной, метода,
     * перечисления и тд
     *
     * Модификатор указывается первым.
     * Виды модификаторов:
     * 1) private - "виден" только внутри этого класса
     * 2) без модификатора - "виден" внутри этого класса и внутри пакета
     * 3) protected - "виден" внутри этого класса, внутри пакета
     * и в классах-наследниках
     * 4) public - "виден" везде
     *
     * Согласно принципу инкапсуляции, класс должен быть закрыт для доступа
     * снаружи, кроме отдельных методов "окошек" для общения с внешним миром
     */

    /**
     * В качестве переменных класса могут выступать и ссылки на объекты
     * То есть, объекты могут содержать ссылки на другие объекты
     */
    private Time startTime;
    private Time endTime;

    /** Далее идет конструктор(-ы) */

    /**
     * Конструктор - это специальный метод
     * он вызывается во время выполнения оператора new
     * который выделяет место в куче под объект.
     * В методе обычно инициализируют переменные, объекты,
     * можно вызвать какой-то метод.
     * Когда вы пишете new Object() - скобки () - это как раз
     * указание на вызов конструктора.
     * Если конструктор с параметрами, то вы в скобках указываете значения
     * параметров.
     *
     * В данном случае, у нас конструктор имеет два параметра
     * то создание объекта этого класса будет иметь такой вид
     * Time time1;
     * Time time2;
     * ...
     * TimeInterval interval = new TimeInterval(time1,time2);
     *
     * Мы видим, что мы должны передать в наш конструктор два объекта класса Time,
     * т.к. у него два входных аргумента. Если мы создадим объект так:
     * TimeInterval interval = new TimeInterval();
     * то компилятор выдаст ошибку, т.к. наш конструктор ТРЕБУЕТ два параметра
     */
    public TimeInterval(Time startTime, Time endTime) {
        if (!setTimeInterval(startTime,endTime)) {
            this.startTime = Time.valueOf(LocalTime.now());
            this.endTime = Time.valueOf(LocalTime.now());
        }
    }

    /** Далее идут методы */

    /**
     * Переменные в 99.9% случаях объявляются как private.
```

```
     * Чтобы дать возможность получить значение переменной или
     * изменить его извне, используются определенные методы класса.
     * Они называются "геттеры" и "сеттеры"
     */

    /**
     * Это метод-геттер.
     * Он позволяет извне получить значение переменной
     *
     * @return значение начальной точки интервала
     */
    public Time getStartTime() {
        return startTime;
    }

    /**
     * Это метод-сеттер, он позволяет установить новое значение для переменной
     * startTime
     * Обратите внимание на использование ключевого слова this.
     * Оно позволяет разрешить конфликт, когда локальная переменная метода
     * и поле класса могут иметь одинаковое имя
     *
     * @param startTime новое значение начальной точки интервала.
     */
    public void setStartTime(Time startTime) {
        this.startTime = startTime;
    }

    /**
     * Вы должны понимать, когда стоит писать геттер и сеттер для полей класса.
     * Тяжело составить какой-то набор правил, но вы должны это понимать
     * исходя из логики того, что класс описывает.
     *
     * В данном случае, класс описывает временной интервал.
     * Временной интервал может иметь различные применения, всё зависит
     * от того, что вы пишете, но в большинстве случаев, временной интервал
     * не может быть отрицательным.
     * Поэтому, если мы напишем отдельно сеттеры для начальной и конечной точки,
     * то логика этого класса может нарушиться, т.к.
     * начальная точка интервала может оказаться "позже" чем конечная точка.
     * Таким образом, кто-то извне может нарушить логику класса.
     * Это называют "утечкой логики".
     * В данном случае, если мы хотим установить значения и не нарушить
     * логику класса, можно написать примерно такой метод.
     *
     * С таким методом мы сразу можем проверить, не нарушается ли логика
     * класса и предотвратить ее нарушение извне
     */
    public boolean setTimeInterval(Time startTime, Time endTime) {
        /**
         * Обратите внимание, что метод возвращает boolean
         * как результат "успешности" операции установки интервала.
         * Многие такие методы имеют boolean либо int как возвращаемый тип,
         * чтобы вернуть "успех\неуспех" действия либо вернуть код ошибки.
         *
         * Например, метод который пишет в файл может возвращать количество
         * записанных байт либо -1 если записать в файл не удалось
         */
        if (!endTime.before(startTime)) {
            this.startTime = startTime;
            this.endTime = endTime;
            return true;
```

```
        } else {
            /**
             * Если endTime раньше, чем startTime, то мы ничего
             * не записываем и возвращаем false, чтобы вызывающий объект
             * извне мог знать, что изменение значений было неудачным
             */
            return false;
        }
    }
}
```

Попробуем написать бинарное дерево с помощью классов

```
package ua.opu.structures;

/**
 * Класс описывает бинарное дерево.
 * Операции с деревом:
 * - вставка элемента
 * - обход дерева (три вида)
 * - удаление элемента
 */
public class BinaryTree {

    /** Корневая вершина */
    private TreeNode rootNode;

    /**
     * Так как дерево, в принципе, может быть пустым,
     * то конструктор у нас пустой
     */
    public BinaryTree() {
    }

    /**
     * Добавление вершины.
     * Если вы не забыли, алгоритм добавления вершины такой:
     * 1. Если корневая вершина пустая - добавляемая вершина становится корневой
     * иначе,
     *
     * Текущая вершина: C - корневая вершина;
     * Добавляемая вершина: N.
     *
     * 2. пока C != null {
     *      если (C.key >= N.key) {C = C.leftChild}
     *      иначе C = C.rightChild
     * }
     * 3. C = N
     *
     * @param key ключ вершины
     * @param value значение вершины
     */
    public void addNode(int key, String value) {
        if (rootNode == null) {
            // Если узел пустой - добавляемая вершина
            // становится узлом
            rootNode = new TreeNode(key, value);
        } else {
            // Текущая вершина - узел
            TreeNode currentNode = rootNode;
```

```
            while (currentNode !=null) {
                if (currentNode.getKey() > key) {
                    // Если левый сын не пустой, тогда спускаемся вниз
                    // если пустой - тогда левый сын становится новым узлом
                    if (currentNode.getLeftChild()!=null) {
                        currentNode = currentNode.getLeftChild();
                    } else {
                        currentNode.setLeftChild(new TreeNode(key,value));
                        break;
                    }
                } else {
                    if (currentNode.getRightChild()!=null) {
                        currentNode = currentNode.getRightChild();
                    } else {
                        currentNode.setRightChild(new TreeNode(key,value));
                        break;
                    }
                }
            }
        }
    }

    public void traverseTree(TraverseOrder order) {
        if (order == TraverseOrder.InOrder) {
            inOrderTraverse(rootNode);
        } else if (order == TraverseOrder.PreOrder) {
            preOrderTraverse(rootNode);
        } else if (order == TraverseOrder.PostOrder) {
            postOrderTraverse(rootNode);
        }
     }

    /**
     * Обратите внимание на следующий важный момент.
     * Чтобы дать внешним объектам возможность обходить дерево
     * мы сделали специальный метод traverseTree, который принимает на вход
     * перечисление в виде порядка обхода,
     * хотя мы могли просто сделать публичными соответствующие методы обхода.
     *
     * Почему же мы написали отдельный метод?
     * Потому что рекурсивный порядок обхода требует, чтобы в качестве параметра мы
указали
     * корневую вершину. Соответственно, внешнему объекту нужно взять эту вершину
     * и вообще появляется информация о том, что есть еще какая-то корневая вершина, а
это
     * ему знать необязательно.
     *
     * Представьте, что ваш класс бинарное дерево предоставляет услуги другим объектам,
     * и вы не должны "грузить" другие объекты лишней информацией, они просто должны
получить
     * свою услугу - обход дерева.
     * В данном случае, наличие корневой вершины - это внутренняя логика работы
     * бинарного дерева и она не должна выставляться наружу.
     *
     */

    private void inOrderTraverse(TreeNode node){
        if (node == null) { return; }
        inOrderTraverse(node.getLeftChild());
        System.out.println(node.toString());
        inOrderTraverse(node.getRightChild());
    }
```

```
    private void preOrderTraverse(TreeNode node){
        if (node == null) { return; }
        System.out.println(node.toString());
        preOrderTraverse(node.getLeftChild());
        preOrderTraverse(node.getRightChild());
    }

    private void postOrderTraverse(TreeNode node){
        if (node == null) { return; }
        postOrderTraverse(node.getLeftChild());
        postOrderTraverse(node.getRightChild());
        System.out.println(node.toString());
    }

    /**
     * Удаление узла. При удалении у нас есть три варианта:
     * 1. У узла нет потомков, тогда просто удаляем
     * 2. У узла есть один из потомков - соединяем потомка с родителем
     * 3. У узла есть оба потомка - берем или предыдущий или последующий элемент,
     * перемещаем его на место удаляемого, применяем удаление к перемещаемому узлу
     *
     * @param key ключ узла, который нужно удалить
     * @return успешность процедуры удаления (false если такого ключа нет в дереве)
     */
    public boolean deleteNode(int key) {
        boolean success = true;

        // ... А вот тут пишите сами

        return success;
    }

    /**
     * Это перечисление.
     * Это такой специальный тип данных,
     * он позволяет выбрать одно из нескольких значений
     */
    public enum TraverseOrder{
        InOrder,
        PreOrder,
        PostOrder
    }
}
```

```
package ua.opu.structures;

/**
 * Класс описывает узел бинарного дерева.
 * Узел дерева должен иметь:
 * - ключ (int)
 * - значение (допустим, у нас будет String)
 * - ссылка на левого потомка (объект класса TreeNode)
 * - ссылка на правого потомка (объект класса TreeNode)
 */
public class TreeNode {

    private int key;
    private String value;
```

```
    private TreeNode leftChild;
    private TreeNode rightChild;

    /**
     * Так как при инициализации узла, у нас
     * вряд ли будет ссылка на дочерние узла,
     * то в конструкторе мы назначим им null и
     * предусмотрим геттер и сеттер для них
     */
    public TreeNode(int key, String value) {
        this.key = key;
        this.value = value;
        this.leftChild = null;
        this.rightChild = null;
    }

    /** Геттеры и сеттеры */
    public TreeNode getLeftChild() {
        return leftChild;
    }
    public void setLeftChild(TreeNode leftChild) {
        this.leftChild = leftChild;
    }

    public TreeNode getRightChild() {
        return rightChild;
    }
    public void setRightChild(TreeNode rightChild) {
        this.rightChild = rightChild;
    }

    public int getKey() {
        return key;
    }

    public String getValue() {
        return value;
    }

    public String toString() {
        return new String ("Ключ: " + key + ". Значение: " + value);
    }
}
```

Выведем на экран результаты работы

```
package ua.opu;

import ua.opu.structures.BinaryTree;

import java.sql.Time;

public class Main {

    public static void main(String[] args) {

        Time time1 = new Time(2,5,26);
        Time time2 = new Time(2,6,79);
```

```
        TimeInterval interval = new TimeInterval(time1,time2);
        System.out.println(interval.getTimeInterval());

        BinaryTree tree = new BinaryTree();
        tree.addNode(50,"root!");
        tree.addNode(45,"1");
        tree.addNode(47,"2");
        tree.addNode(39,"3");
        tree.addNode(62,"4");
        tree.addNode(69,"5");

        System.out.println("Прямой порядок");
        tree.traverseTree(BinaryTree.TraverseOrder.InOrder);
        System.out.println("Симметричный порядок");
        tree.traverseTree(BinaryTree.TraverseOrder.PreOrder);
        System.out.println("Обратный порядок");
        tree.traverseTree(BinaryTree.TraverseOrder.PostOrder);
    }
}
```

ЗАДАНИЯ НА ЛАБОРАТОРНУЮ РАБОТУ:

1. Создайте класс Employee который содержит три переменные – имя, фамилия и месячная зарплата. Создайте конструктор который инициализирует три этих переменные. Создайте геттеры и сеттеры для каждой переменной. Сделайте проверку при установке зарплаты на отрицательное значение. Создайте два объекта и выведите их годовую зарплату. Поднимите зарплату каждому объекту на 10% и снова выведите значения.

2. Реализуйте метод удаления узла из бинарного дерева в существующем классе, который был дан вам в «Ходе работы».

3. Реализуйте структуру данных, в классе Main предусмотрите примеры работы со структурой, добавление, выборку элементов, с выводом на консоль.
а) Если ваша фамилия заканчивается на гласную – реализуйте двусвязную очередь и следующие операции:

- pushBack – добавление в конец очереди;
- pushFront – добавление в начало очереди;
- popBack – выборка с конца очереди;
- popFront – выборка с начала очереди;
- проверка наличия элементов;
- очистка

б) Если ваша фамилия заканчивается на согласную – реализуйте кольцевой буфер (16 ячеек) и следующие операции:

- добавление элемента;
- выборка элемента;
- проверка наличия элементов;
- проверка количества свободных ячеек;
- вывод конечных элементов
- очистка