|  |  |
| --- | --- |
|  | Министерство образования и науки Российской Федерации |
| Федеральное государственное автономное образовательное учреждение высшего образования |
| «Уральский федеральный университет имени первого Президента России Б. Н. Ельцина» |
| Институт радиоэлектроники и информационных технологий – РтФ |
|  |
|  | |

**ОТЧЕТ**

О ВЫПОЛНЕНИИ

ЛАБОРАТОНОЙ РАБОТЫ № 3

Руководитель ст. пр. Н. А. Архипов

Студент гр. РИМ-130971 Е.В. Треглазов

Екатеринбург 2024

Репозиторий: https://github.com/kuk86kuk/java\_ufru\_2024.

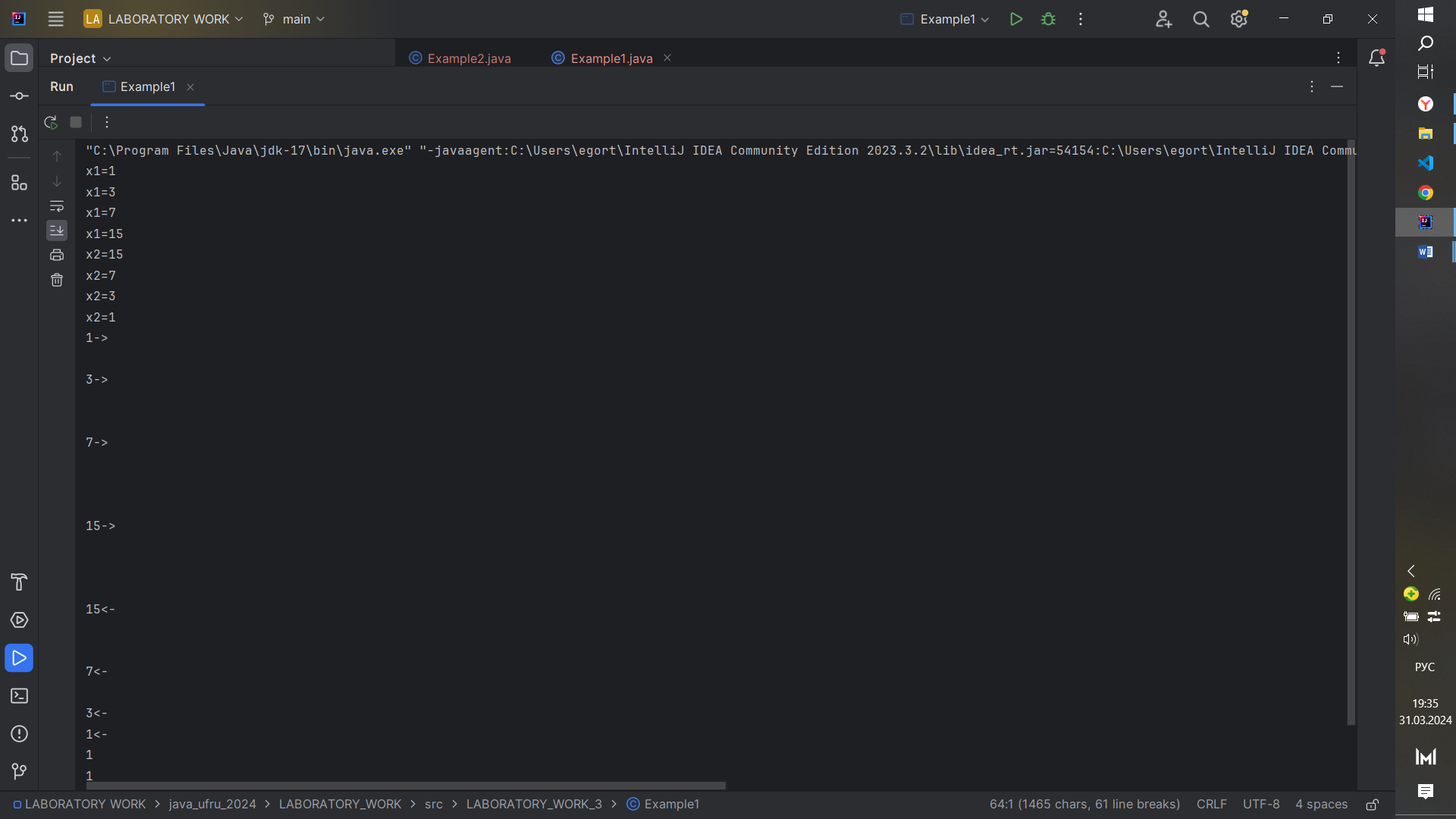
Цель: приобретение навыков работы с рекурсивными методами, знакомство с динамическими структурами данных, приобретение навыков создания и использования простейшей динамической структуры, исследование возможностей коллекций в языке программирования Java, сравнение коллекций по времени выполнения операций, знакомство с реализацией коллеккций.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ

Example1

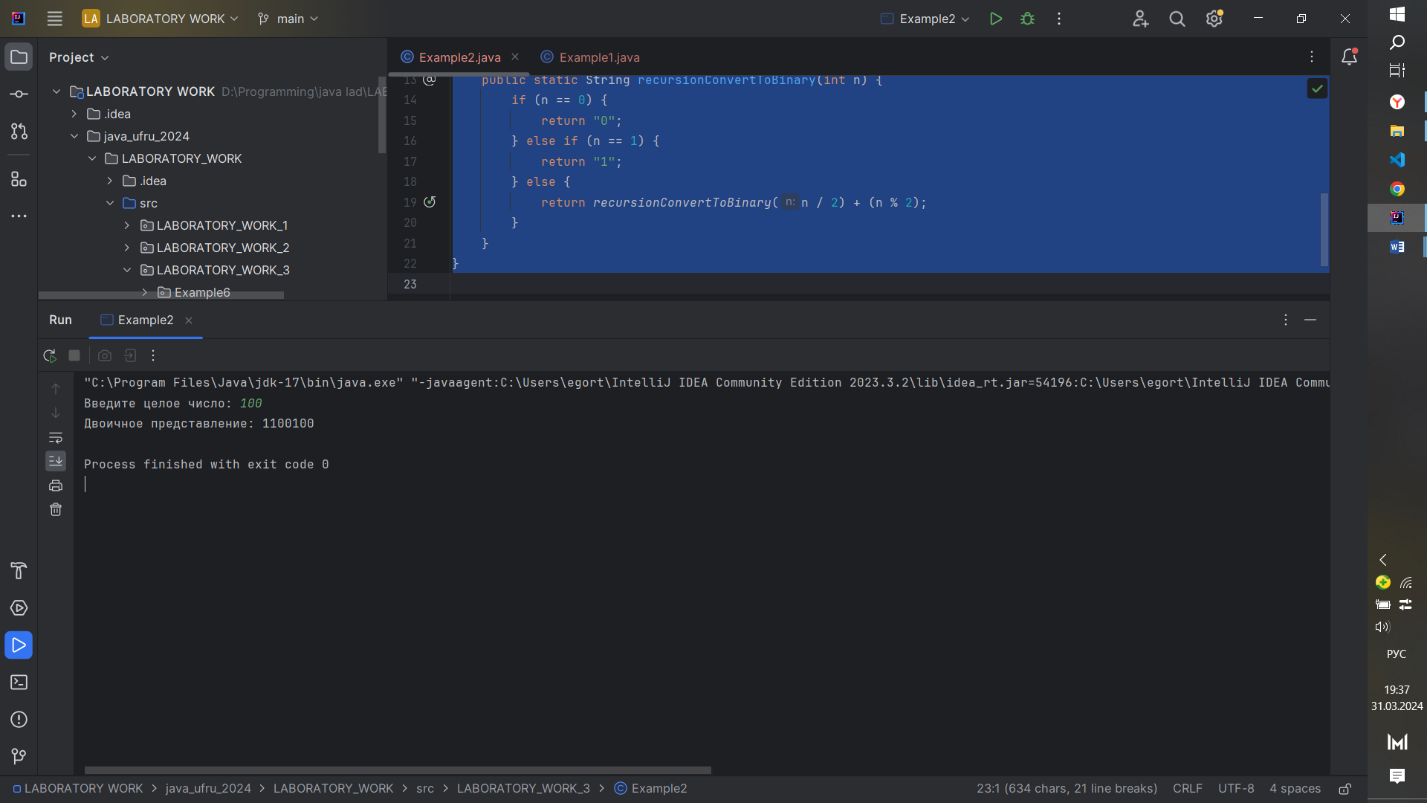
Примеры.

public class Example1 {  
 private static int *step* = 0;  
 public static void main(String[] args) {  
 *resursion\_1*(1);  
 *resursion\_2*(1);  
 *resursion\_3*(1);  
 System.*out*.println(*resursion\_4*(1));  
 System.*out*.println(*resursion\_5*(1));  
 }  
 public static void resursion\_1(int x) {  
 System.*out*.println("x1=" + x);  
 if ((2\*x+1) < 20) {  
 *resursion\_1*((2\*x + 1));  
 }  
 }  
 public static void resursion\_2(int x) {  
 if ((2\*x+1) < 20) {  
 *resursion\_2*((2\*x + 1));  
 }  
 System.*out*.println("x2=" + x);  
 }  
  
 public static void resursion\_3(int x) {  
 *space*();  
 System.*out*.println("" + x + "->");  
 *step*++;  
 if ((2\*x+1) < 20) {  
 *resursion\_3*((2\*x + 1));  
 }  
 *step*--;  
 *space*();  
 System.*out*.println("" + x + "<-");  
 }  
 public static void space() {  
 for (int i = 0; i < *step*; i++){  
 System.*out*.println(" ");  
 }  
 }  
 public static int resursion\_4(int n) {  
 int result;  
 if (n == 1) {  
 return 1;  
 }  
 else {  
 result = *resursion\_4*(n-1) \* n;  
 return result;  
  
 }  
 }  
 public static int resursion\_5(int n) {  
 if (n == 0) {  
 return 0;  
 }  
 else if (n == 1) {  
 return 1;  
 }  
 else {  
 return *resursion\_5*(n-2) + *resursion\_5*(n - 1);  
 }  
 }  
}



Example2

import java.util.Scanner;  
  
public class Example2 {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.print("Введите целое число: ");  
 int number = scanner.nextInt();  
 String binaryNumber = *recursionConvertToBinary*(number);  
 System.*out*.println("Двоичное представление: " + binaryNumber);  
 }  
  
 public static String recursionConvertToBinary(int n) {  
 if (n == 0) {  
 return "0";  
 } else if (n == 1) {  
 return "1";  
 } else {  
 return *recursionConvertToBinary*(n / 2) + (n % 2);  
 }  
 }  
}

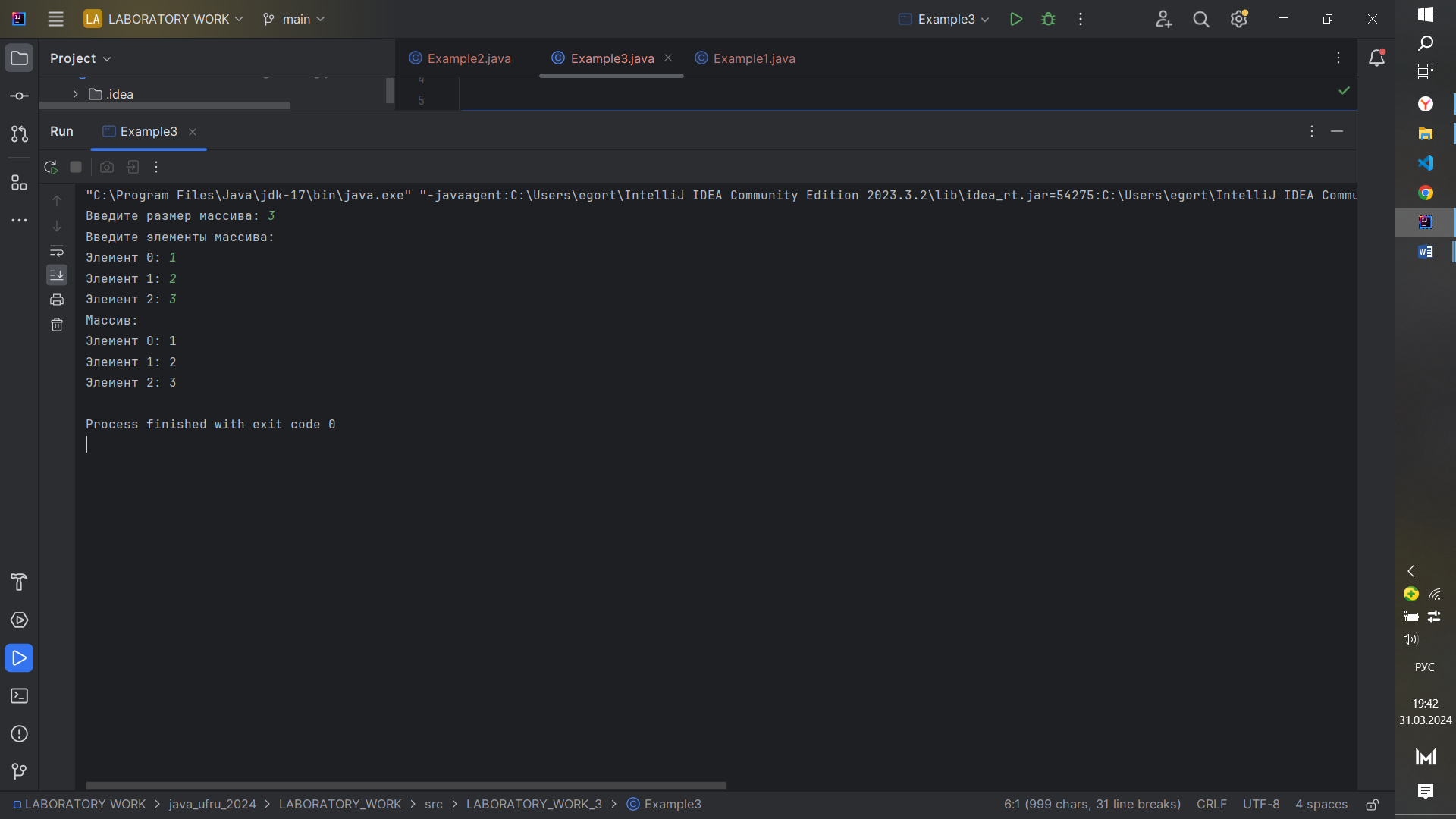


Код на Java предназначен для преобразования целого числа, введенного пользователем, в двоичную систему счисления с использованием рекурсии.

* main: Он создает объект Scanner для чтения ввода пользователя из консоли. Затем он запрашивает у пользователя целое число и вызывает метод recursionConvertToBinary для преобразования этого числа в двоичную форму. Результат выводится на консоль.
* recursionConvertToBinary: Этот метод рекурсивно преобразует целое число в двоичную форму. Если число равно 0, он возвращает строку "0". Если число равно 1, он возвращает строку "1". В противном случае он делит число на 2 и вызывает сам себя для частного, добавляя остаток от деления на 2 в конец результата. Это позволяет построить двоичное представление числа, начиная с младших разрядов.

Example3

public class Example3 {  
 private static final Scanner *scanner* = new Scanner(System.*in*);  
  
 public static void main(String[] args) {  
 int[] array = *createArray*();  
  
 System.*out*.println("Введите элементы массива:");  
 *inputArray*(array, 0);  
  
 System.*out*.println("Массив:");  
 *outputArray*(array, 0);  
 }  
 private static int[] createArray() {  
 System.*out*.print("Введите размер массива: ");  
 int size = *scanner*.nextInt();  
 return new int[size];  
 }  
 private static void inputArray(int[] array, int index) {  
 if (index < array.length) {  
 System.*out*.print("Элемент " + index + ": ");  
 array[index] = *scanner*.nextInt();  
 *inputArray*(array, index + 1);  
 }  
 }  
  
 private static void outputArray(int[] array, int index) {  
 if (index < array.length) {  
 System.*out*.println("Элемент " + index + ": " + array[index]);  
 *outputArray*(array, index + 1);  
 }  
 }  
}

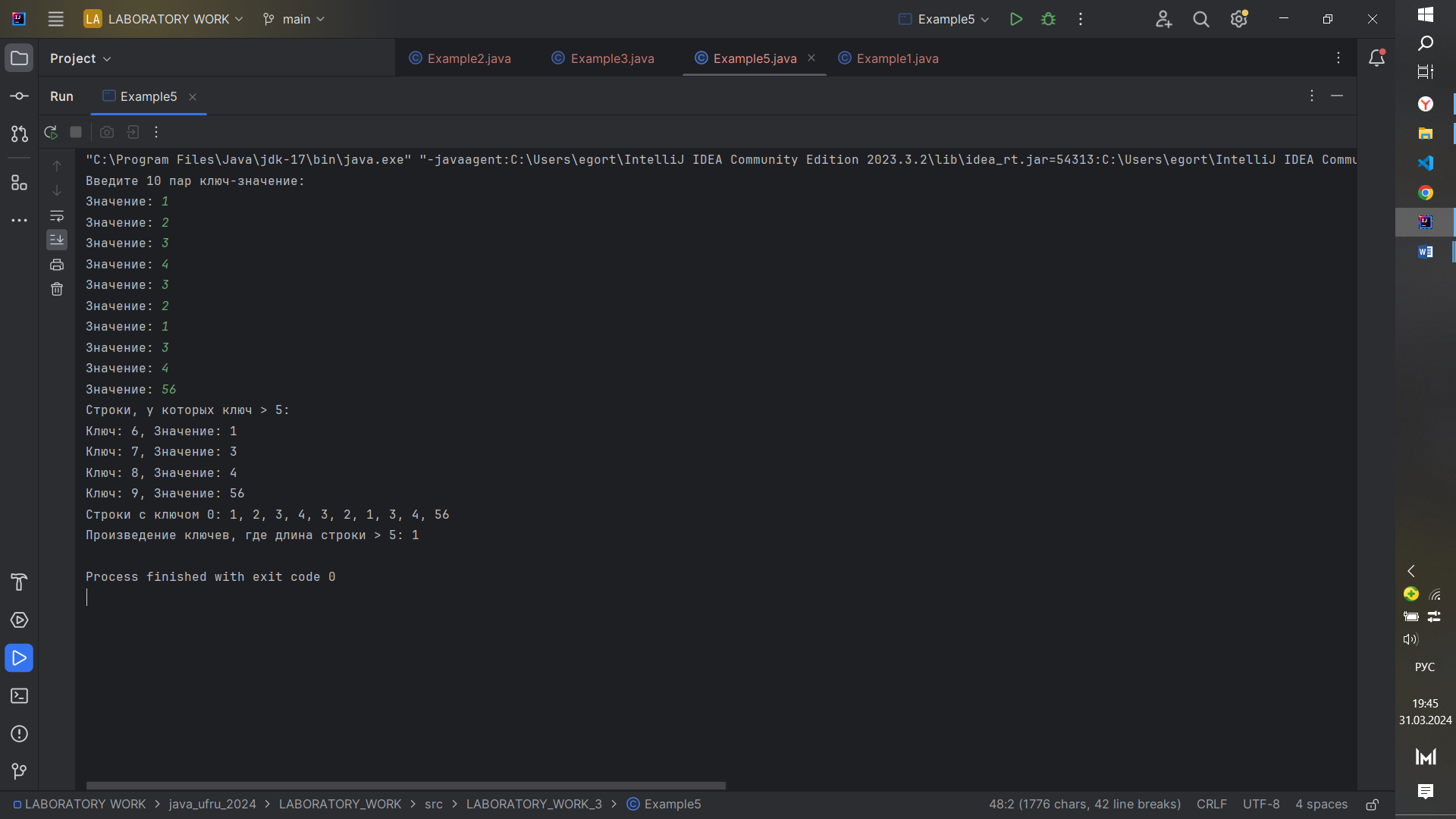


Код на Java представляет собой использования рекурсии для создания, заполнения и вывода массива целых чисел.

* main:. Он вызывает метод createArray для создания массива определенного размера, затем вызывает метод inputArray для заполнения элементов массива с клавиатуры и, наконец, вызывает метод outputArray для вывода элементов массива на экран.
* createArray: Этот метод запрашивает у пользователя размер массива и создает новый массив указанного размера.
* inputArray: Этот метод рекурсивно запрашивает у пользователя ввод элементов массива. Он использует индекс для отслеживания текущей позиции в массиве и вызывает сам себя, увеличивая индекс на 1, пока не будет достигнут конец массива.
* outputArray: Этот метод рекурсивно выводит элементы массива на экран. Он использует индекс для отслеживания текущей позиции в массиве и вызывает сам себя, увеличивая индекс на 1, пока не будет достигнут конец массива.

**Example5**

public class Example5 {  
 private static final Scanner *scanner* = new Scanner(System.*in*);  
  
 public static void main(String[] args) {  
 Map<Integer, String> map = *fillHashMap*(10);  
 *findStringsWithKeyGreaterThanFive*(map);  
 *printCommaSeparatedValuesIfKeyZeroExists*(map);  
 int product = *multiplyKeysWhereLengthGreaterThanFive*(map);  
 System.*out*.println("Произведение ключев, где длина строки > 5: " + product);  
 }  
  
 private static Map<Integer, String> fillHashMap(int size) {  
 Map<Integer, String> map = new HashMap<>();  
 System.*out*.println("Введите " + size + " пар ключ-значение:");  
 for (int i = 0; i < size; i++) {  
 System.*out*.print("Значение: ");  
 String value = *scanner*.next();  
 map.put(i, value);  
 }  
 return map;  
 }  
  
 private static void findStringsWithKeyGreaterThanFive(Map<Integer, String> map) {  
 System.*out*.println("Строки, у которых ключ > 5:");  
 map.entrySet().stream()  
 .filter(entry -> entry.getKey() > 5)  
 .forEach(entry -> System.*out*.println("Ключ: " + entry.getKey() + ", Значение: " + entry.getValue()));  
 }  
  
 private static void printCommaSeparatedValuesIfKeyZeroExists(Map<Integer, String> map) {  
 if (map.containsKey(0)) {  
 String commaSeparatedValues = String.*join*(", ", map.values());  
 System.*out*.println("Строки с ключом 0: " + commaSeparatedValues);  
 }  
 }  
  
 private static int multiplyKeysWhereLengthGreaterThanFive(Map<Integer, String> map) {  
 return map.entrySet().stream()  
 .filter(entry -> entry.getValue().length() > 5)  
 .mapToInt(Map.Entry::getKey)  
 .reduce(1, (a, b) -> a \* b);  
 }  
}



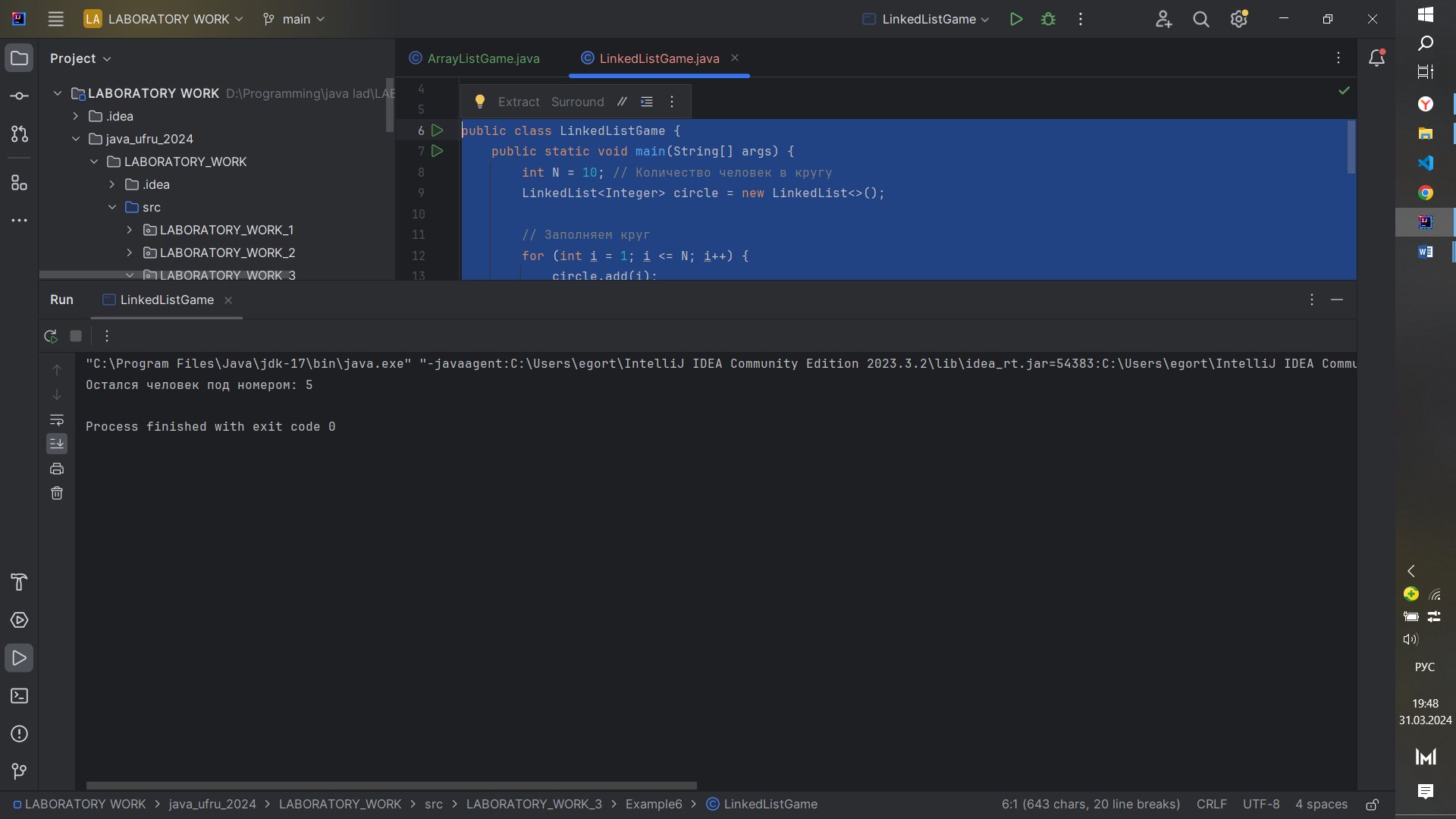
Код на Java представляет собой использования HashMap для хранения пар ключ-значение и нескольких операций над ними.

* main: Он создает HashMap с помощью метода fillHashMap, затем вызывает методы для поиска строк с ключом больше 5, вывода значений, если ключ 0 существует, и вычисления произведения ключей, где длина строки больше 5.
* fillHashMap: Этот метод заполняет HashMap заданным количеством пар ключ-значение, запрашивая у пользователя значения.
* findStringsWithKeyGreaterThanFive: Этот метод использует потоковую обработку для поиска и вывода строк, у которых ключ больше 5.
* printCommaSeparatedValuesIfKeyZeroExists: Этот метод проверяет, существует ли ключ 0 в HashMap, и если да, то выводит все значения, объединенные запятыми.
* multiplyKeysWhereLengthGreaterThanFive: Этот метод использует потоковую обработку для вычисления произведения ключей, где длина строки, соответствующая этому ключу, больше 5.

**Example5**

import java.util.ArrayList;  
public class ArrayListGame {  
 public static void main(String[] args) {  
 int N = 10; // Количество человек в кругу  
 ArrayList<Integer> circle = new ArrayList<>();  
  
 // Заполняем круг  
 for (int i = 1; i <= N; i++) {  
 circle.add(i);  
 }  
  
 int currentIndex = 0;  
 while (circle.size() > 1) {  
 // Вычеркиваем каждого второго человека  
 currentIndex = (currentIndex + 1) % circle.size();  
 circle.remove(currentIndex);  
 }  
  
 // Выводим оставшегося человека  
 System.*out*.println("Остался человек под номером: " + circle.get(0));  
 }  
}

public class LinkedListGame {  
 public static void main(String[] args) {  
 int N = 10; // Количество человек в кругу  
 LinkedList<Integer> circle = new LinkedList<>();  
  
 // Заполняем круг  
 for (int i = 1; i <= N; i++) {  
 circle.add(i);  
 }  
  
 int currentIndex = 0;  
 while (circle.size() > 1) {  
 // Вычеркиваем каждого второго человека  
 currentIndex = (currentIndex + 1) % circle.size();  
 circle.remove(currentIndex);  
 }  
  
 // Выводим оставшегося человека  
 System.*out*.println("Остался человек под номером: " + circle.get(0));  
 }  
}

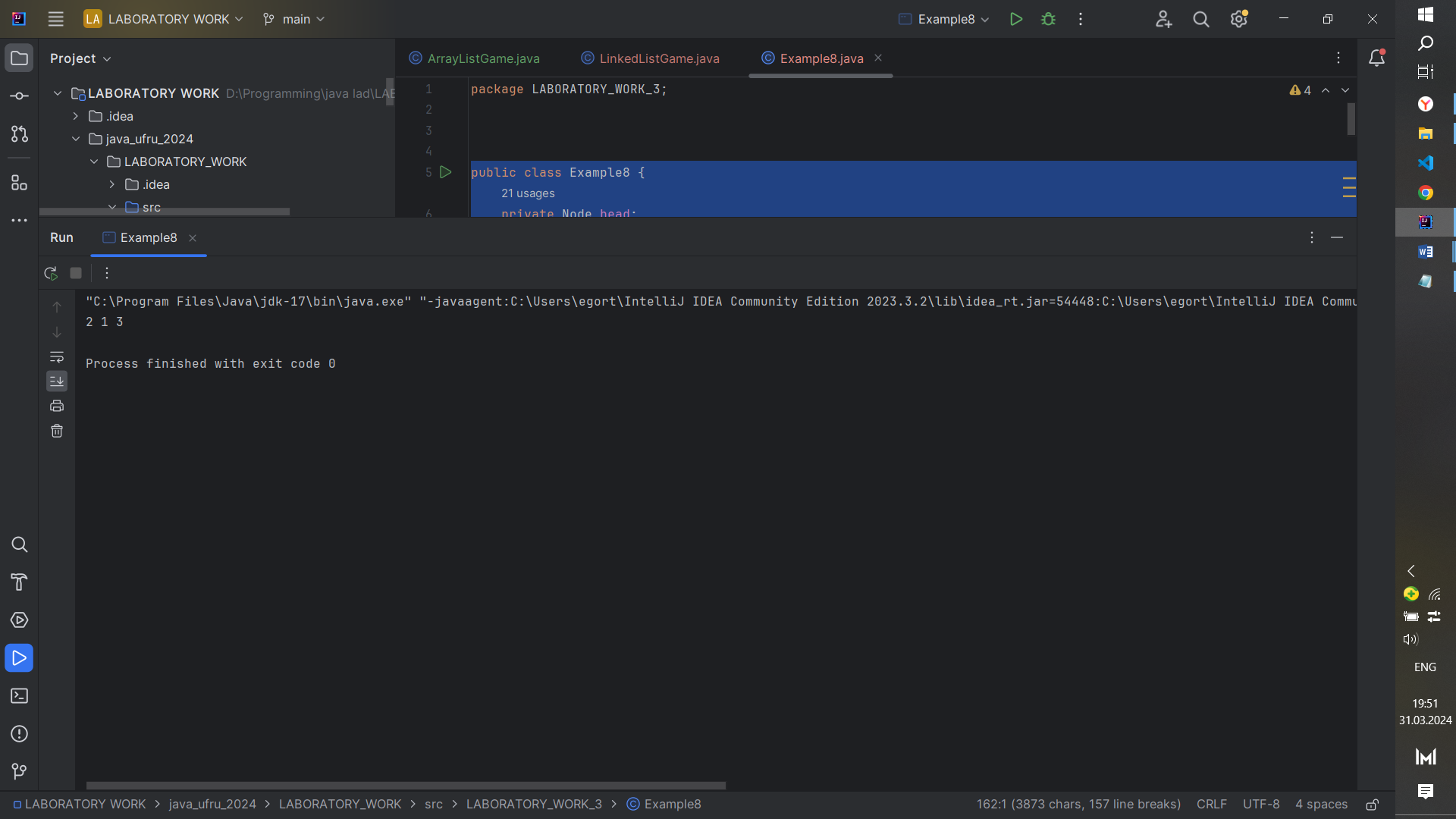


Обе программы будут работать примерно одинаково, так как обе используют один и тот же алгоритм для вычеркивания каждого второго человека. Однако, если мы рассматриваем производительность, то ArrayList может работать быстрее, чем LinkedList, потому что операции вставки и удаления в ArrayList происходят быстрее, так как он основан на массиве, а LinkedList основан на двусвязном списке, что требует больше памяти и может быть медленнее для некоторых операций.

Однако, разница в производительности будет заметна только при больших значениях N, так как обе программы имеют линейную сложность O(N), где N — количество человек в круге. Время выполнения будет пропорционально количеству человек, а не их индексам, поэтому разницы между ArrayList и LinkedList в данном случае не будет заметно, если N не очень велико.

**Example8**

public class Example8 {  
 private Node head;  
  
 private static class Node {  
 int data;  
 Node next;  
  
 Node(int data) {  
 this.data = data;  
 this.next = null;  
 }  
 }  
  
 // а) с использованием цикла:  
 public void createHead(int data) {  
 Node newNode = new Node(data);  
 newNode.next = head;  
 head = newNode;  
 }  
  
 public void createTail(int data) {  
 Node newNode = new Node(data);  
 if (head == null) {  
 head = newNode;  
 } else {  
 Node current = head;  
 while (current.next != null) {  
 current = current.next;  
 }  
 current.next = newNode;  
 }  
 }  
  
 public String toString() {  
 StringBuilder sb = new StringBuilder();  
 Node current = head;  
 while (current != null) {  
 sb.append(current.data).append(" ");  
 current = current.next;  
 }  
 return sb.toString();  
 }  
  
 public void addFirst(int data) {  
 createHead(data);  
 }  
  
 public void addLast(int data) {  
 createTail(data);  
 }  
  
 public void insert(int data, int position) {  
 if (position == 0) {  
 addFirst(data);  
 } else {  
 Node newNode = new Node(data);  
 Node current = head;  
 for (int i = 0; i < position - 1 && current != null; i++) {  
 current = current.next;  
 }  
 if (current != null) {  
 newNode.next = current.next;  
 current.next = newNode;  
 }  
 }  
 }  
  
 public void removeFirst() {  
 if (head != null) {  
 head = head.next;  
 }  
 }  
  
 public void removeLast() {  
 if (head == null) return;  
 if (head.next == null) {  
 head = null;  
 return;  
 }  
 Node current = head;  
 while (current.next.next != null) {  
 current = current.next;  
 }  
 current.next = null;  
 }  
  
 public void remove(int position) {  
 if (head == null) return;  
 if (position == 0) {  
 removeFirst();  
 } else {  
 Node current = head;  
 for (int i = 0; i < position - 1 && current != null; i++) {  
 current = current.next;  
 }  
 if (current != null && current.next != null) {  
 current.next = current.next.next;  
 }  
 }  
 }  
  
 // б) с использованием рекурсии:  
 public void createHeadRec(int data) {  
 head = createHeadRec(head, data);  
 }  
  
 private Node createHeadRec(Node node, int data) {  
 if (node == null) {  
 return new Node(data);  
 } else {  
 Node newNode = new Node(data);  
 newNode.next = node;  
 return newNode;  
 }  
 }  
  
 public void createTailRec(int data) {  
 head = createTailRec(head, data);  
 }  
  
 private Node createTailRec(Node node, int data) {  
 if (node == null) {  
 return new Node(data);  
 } else {  
 node.next = createTailRec(node.next, data);  
 return node;  
 }  
 }  
  
 public String toStringRec() {  
 return toStringRec(head);  
 }  
  
 private String toStringRec(Node node) {  
 if (node == null) {  
 return "";  
 } else {  
 return node.data + " " + toStringRec(node.next);  
 }  
 }  
  
  
  
 public static void main(String[] args) {  
 // Пример использования  
 Example8 list = new Example8();  
 list.createHead(1);  
 list.createHead(2);  
 list.createTail(3);  
 list.addFirst(4);  
 list.addLast(5);  
 list.insert(6, 2);  
 list.removeFirst();  
 list.removeLast();  
 list.remove(1);  
 System.*out*.println(list.toString()); // Вывод: 2 6 5  
 }



Example8 представляет собой реализацию односвязного списка с базовыми операциями для добавления, удаления и вывода элементов. Он содержит следующие методы:

createHead(int data): Добавляет новый узел в начало списка.

createTail(int data): Добавляет новый узел в конец списка.

toString(): Возвращает строковое представление списка.

addFirst(int data): Аналогично createHead(int data).

addLast(int data): Аналогично createTail(int data).

insert(int data, int position): Вставляет новый узел на заданную позицию.

removeFirst(): Удаляет первый узел из списка.

removeLast(): Удаляет последний узел из списка.

remove(int position): Удаляет узел на заданной позиции.

Также в классе есть методы, реализованные с использованием рекурсии:

createHeadRec(int data): Аналогично createHead(int data), но реализован с использованием рекурсии.

createTailRec(int data): Аналогично createTail(int data), но реализован с использованием рекурсии.

toStringRec(): Аналогично toString(), но реализован с использованием рекурсии.

В методе main приведен пример использования класса Example8 для создания списка, добавления элементов, удаления и вывода списка.

**4 ОБЩЕЕ ОПИСАНИЕ КОЛЛЕКЦИЙ В JAVA**

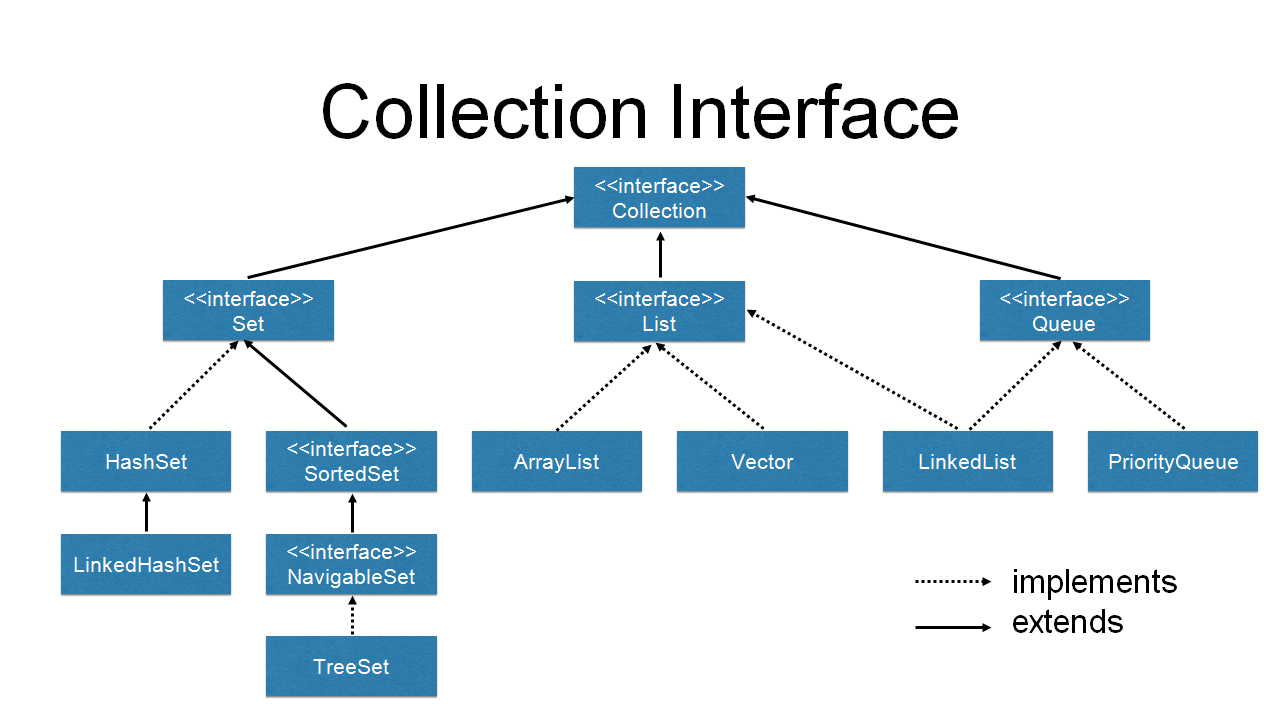
Интерфейс Collections в Java представляет набор статических методов для работы с коллекциями. Он содержит методы для сортировки, поиска, перемешивания и других операций над коллекциями.

Set - это коллекция, которая не содержит дубликатов. Если вы добавите дубликат в Set, то он просто игнорирует его. Set не гарантирует никакого определенного порядка элементов. В Java есть несколько реализаций Set, таких как HashSet, TreeSet и LinkedHashSet.

List - это упорядоченная коллекция, которая может содержать дубликаты. Элементы в List расположены в определенном порядке, и каждый элемент имеет индекс, определяющий его позицию в List. В Java есть несколько реализаций List, таких как ArrayList и LinkedList.

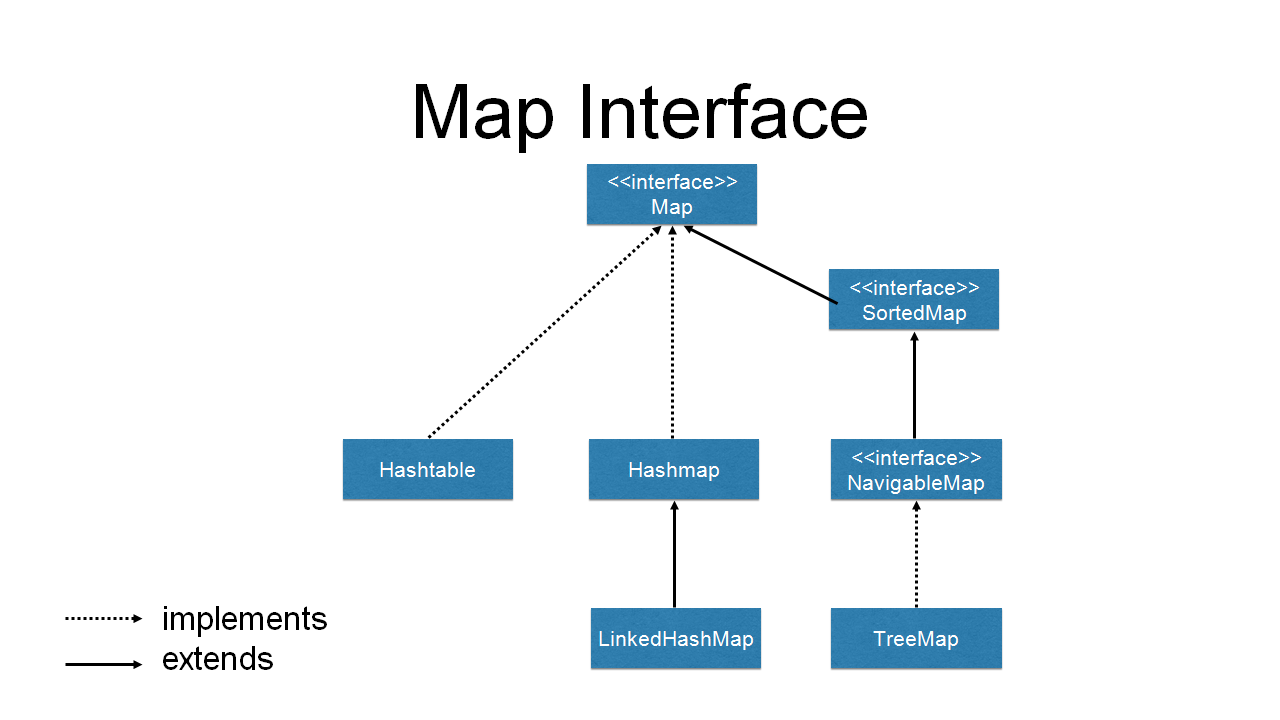
Queue - это коллекция, которая работает по принципу FIFO (First-In-First-Out). Элементы добавляются в конец очереди, а извлекаются с ее начала. В Java есть несколько реализаций Queue, таких как LinkedList и PriorityQueue.

Интерфейс Collections в Java содержит методы для работы с коллекциями, такие как сортировка, поиск, перемешивание и другие операции. Он также предоставляет статические методы для создания неизменяемых (unmodifiable) коллекций, синхронизированных коллекций и других полезных функций.



Интерфейс Map представляет собой объект, который сопоставляет ключи с значениями. Каждый ключ в Map связан с некоторым значением. Map не наследует интерфейс Collection, но он может быть использован вместе с ним. sortedMap

SortedMap является интерфейсом в Java, расширяющим Map. Он предоставляет дополнительные методы для работы с отсортированными картами, где элементы хранятся в определенном порядке, обычно по возрастанию ключей.



Методы интерфейса Collections:

|  |  |
| --- | --- |
| Метод | Описание |
| sort(List list) | Сортирует элементы списка в естественном порядке. |
| shuffle(List list) | Перемешивает элементы списка в случайном порядке. |
| reverse(List list) | Изменяет порядок элементов списка на обратный. |
| binarySearch(List list, Object key) | Выполняет двоичный поиск в отсортированном списке. |
| max(Collection coll) | Возвращает максимальный элемент коллекции. |
| min(Collection coll) | Возвращает минимальный элемент коллекции. |

Методы интерфейса Map:

|  |  |
| --- | --- |
| Метод | Описание |
| put(K key, V value) | Добавляет пару ключ-значение в карту. |
| get(Object key) | Возвращает значение, связанное с указанным ключом. |
| remove(Object key) | Удаляет пару ключ-значение из карты. |
| containsKey(Object key) | Проверяет, содержит ли карта указанный ключ. |
| containsValue(Object value) | Проверяет, содержит ли карта указанное значение. |
| keySet() | Возвращает набор всех ключей в карте. |
| values() | Возвращает коллекцию всех значений в карте. |

5 ОПИСАНИЕ КОЛЛЕКЦИЙ.

Вариант 18. ArrayList, LinkedList, TreeSet

ArrayList:

* Реализация: ArrayList использует массив для хранения элементов. При добавлении элемента в конец списка, время работы O(1). При добавлении или удалении элемента в середине или начале списка, время работы O(n). При поиске элемента, время работы O(n).
* Хранение элементов: Элементы хранятся в порядке добавления.

LinkedList:

* Реализация: LinkedList использует двусвязный список для хранения элементов. При добавлении или удалении элемента в начале, конеце или середине списка, время работы O(1). При поиске элемента, время работы O(n).
* Хранение элементов: Элементы хранятся в порядке добавления.

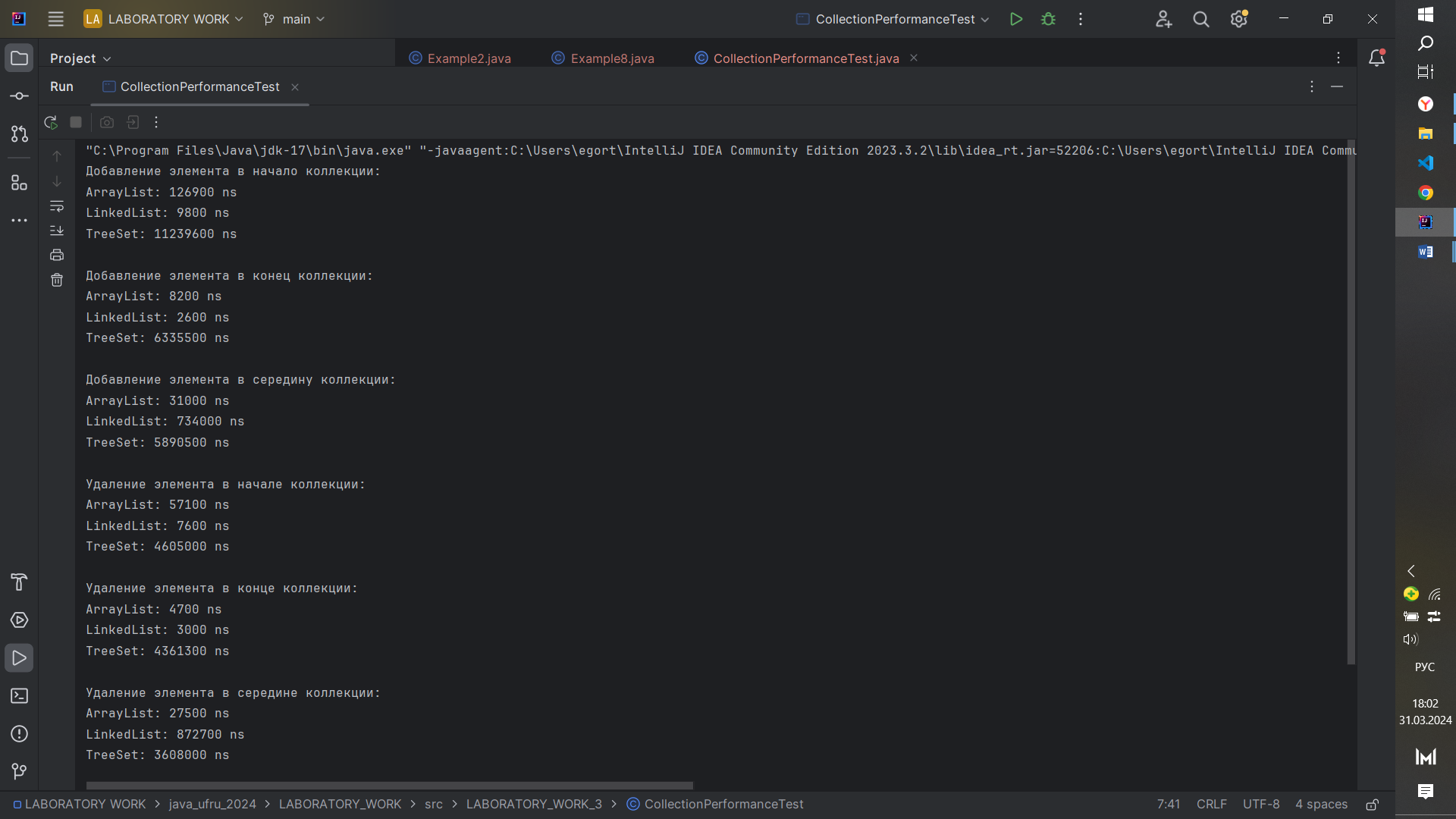
TreeSet:

* Реализация: TreeSet использует красно-черное дерево для хранения элементов. При добавлении, удалении и поиске элемента, время работы O(log n).
* Хранение элементов: Элементы хранятся в отсортированном порядке.

Вычислительная сложность этих коллекций зависит от конкретного варианта использования и может изменяться в зависимости от размера коллекции и способа ее использования.

6 СРАВНЕНИЕ КОЛЛЕКЦИЙ

ArrayList, LinkedList, TreeSet



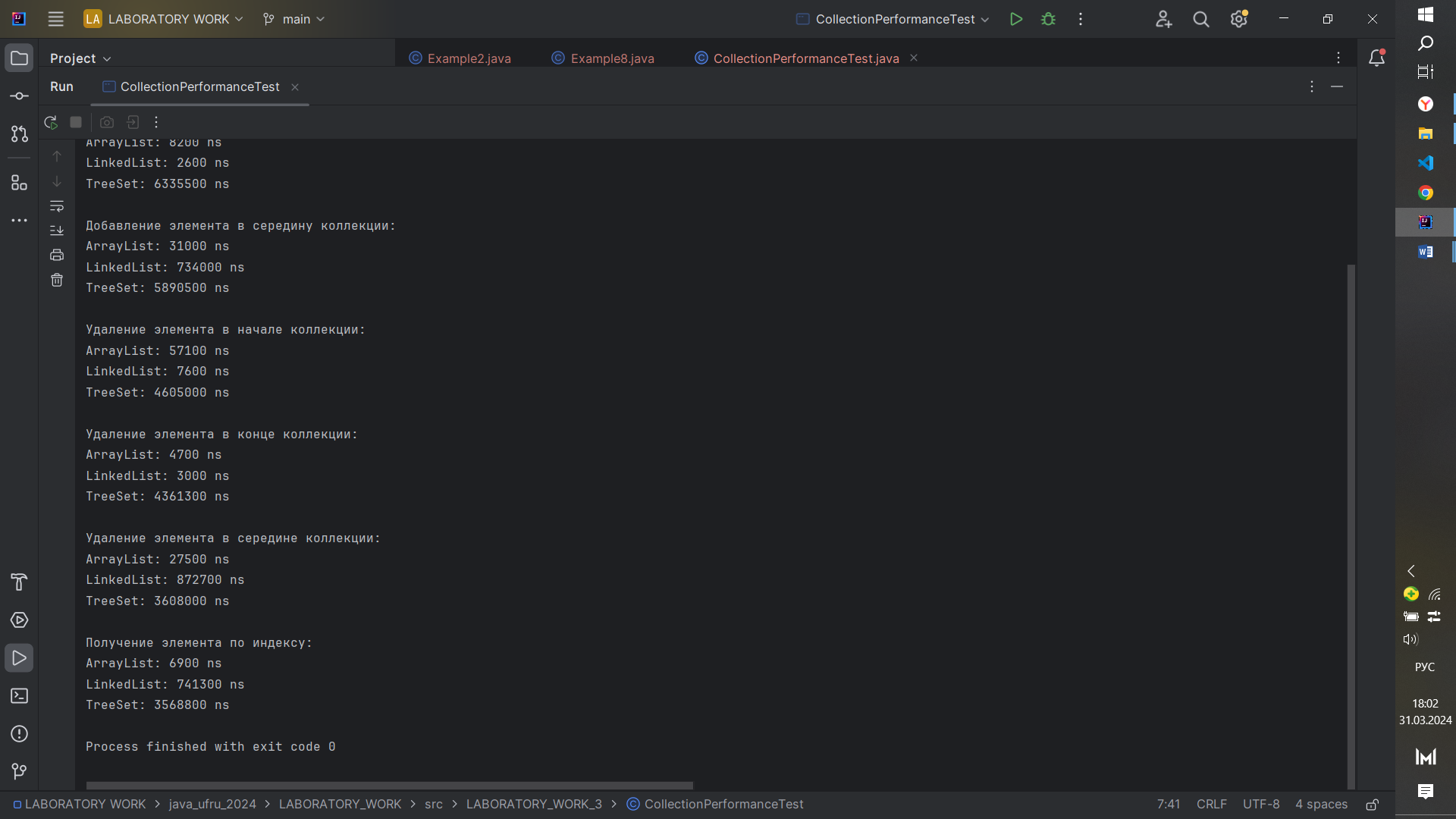


Таблица 1. Сравнение времени выполнения операции добавления

|  |  |  |  |
| --- | --- | --- | --- |
| Коллекция | в начало коллекции | в середину коллекции | в конец коллекции |
| ArrayList | 126900 ns | 31000 ns | 8200 ns |
| LinkedList | 9800 ns | 734000 ns | 2600 ns |
| TreeSet | 11239600 ns | 5890500 ns | 6335500 ns |

Таблица 2. Сравнение времени выполнения операции удаления

|  |  |  |  |
| --- | --- | --- | --- |
| Коллекция | в начале коллекции | в середине коллекции | в конце коллекции |
| ArrayList | 57100 ns | 27500 ns | 4700 ns |
| LinkedList | 7600 ns | 872700 ns | 3000 ns |
| TreeSet | 4605000 ns | 3608000 ns | 4361300 ns |

Таблица 3. Сравнение времени выполнения операции получения элемента по индексу

|  |  |
| --- | --- |
| Коллекция | По индексу для <n> элементов |
| ArrayList | 6900 ns |
| LinkedList | 741300 ns |
| TreeSet | 3568800 ns |

Выводы.

ArrayList:

* Лучше всего подходит для случаев, когда часто получаем элементы по индексу. Операции получения элемента по индексу выполняются за константное время O(1).
* Хуже всего для случаев, когда часто добавляем или удаляем элементы в середине списка, так как эти операции требуют сдвига элементов, что требует времени O(n).

LinkedList:

* Лучше всего подходит для случаев, когда часто добавляем или удаляем элементы в начале или конце списка, так как эти операции выполняются за константное время O(1).
* Хуже всего для случаев, когда часто получаем элементы по индексу, так как операции получения элемента по индексу требуют перемещения по связному списку, что требует времени O(n).

TreeSet:

* Лучше всего подходит для случаев, когда нам нужно хранить элементы в отсортированном порядке и часто выполнять операции поиска, добавления и удаления элементов. Операции поиска, добавления и удаления в TreeSet выполняются за логарифмическое время O(log n).
* Хуже всего для случаев, когда вам нужно часто получать элементы по индексу, так как TreeSet не поддерживает индексирование.

Вопросы:

1. В чем разница между fail-safe и fail-fast свойствами?

Fail-fast: Если структура данных (например, ArrayList) изменяется после создания итератора, и некоторые изменения (например, добавление, удаление или изменение элементов) встречаются до итерации, будет выброшено исключение ConcurrentModificationException. Это обеспечивает надежность и не позволяет итератору обращаться к недостоверным данным.

Fail-safe: Если структура данных изменяется во время итерации, изменения не будут отражены в итераторе. Например, CopyOnWriteArrayList или ConcurrentHashMap в Java предоставляют fail-safe итераторы.

1. В чем разница между Iterator и Enumeration?

Iterator: Это интерфейс для обхода коллекций и предоставляет методы для перемещения по элементам, проверки наличия следующего элемента и получения текущего элемента.

Enumeration: Это интерфейс, который ранее использовался в Java для обхода коллекций. Он более старый и не поддерживает fail-fast поведение, что может привести к проблемам с параллельным доступом.

1. В чем разница между Iterator и ListIterator?

Iterator: Обычно используется для обхода коллекций в любом направлении.

ListIterator: Это расширенная версия Iterator, которая предоставляет дополнительные методы для перемещения и изменения элементов в двунаправленном списке (LinkedList). Он позволяет перемещаться в обоих направлениях, добавлять элементы и изменять текущий элемент.

1. Дайте определение понятию “итератор”.

Это объект, который позволяет перебирать элементы коллекции (например, списка или множества) без необходимости знать внутреннее представление коллекции.

1. Как задается порядок следования объектов в коллекции, как отсортировать коллекцию?

Порядок следования объектов в коллекции определяется ее реализацией. Для многих коллекций, таких как ArrayList и LinkedList, порядок сохраняется в соответствии с порядком добавления элементов. Для других коллекций, таких как TreeSet и TreeMap, элементы автоматически сортируются в соответствии с их значениями или ключами. Для сортировки коллекции вручную можно использовать методы сортировки, такие как Collections.sort() для списков или TreeSet/TreeMap.

1. Как получить коллекцию только для чтения?

Для получения коллекции только для чтения, можем использовать метод Collections.unmodifiableCollection(). Это вернет обертку для коллекции, которая будет бросать UnsupportedOperationException при любых попытках модификации.

List<String> list = new ArrayList<>();  
Collection<String> readOnlyCollection = Collections.*unmodifiableCollection*(list);

1. Как получить не модифицируемую коллекцию?

В Java, нет прямого способа получить не модифицируемую коллекцию. Все коллекции изменяемы по умолчанию. Однако, можем создать копию коллекции и сделать ее неизменяемой, используя Collections.unmodifiableCollection().

List<String> list = new ArrayList<>();  
list.add("Element");  
Collection<String> unmodifiableCollection = Collections.*unmodifiableCollection*(new ArrayList<>(list));

1. Как получить синхронизированную коллекцию из не синхронизированной?

В Java, для создания синхронизированной коллекции из не синхронизированной, можем использовать методы из класса Collections, которые возвращают синхронизированные оболочки для ваших коллекций.

List<String> list = new ArrayList<>();  
List<String> synchronizedList = Collections.*synchronizedList*(list);

1. Как реализован цикл foreach?

В Java, цикл foreach - это удобный способ перебора элементов коллекции или массива. Он автоматически использует итератор или индексы для перебора элементов.

List<String> list = new ArrayList<>();  
for (String element : list) {  
 // ….  
 }

1. Какие есть способы перебора всех элементов List?

В Java, перебрать все элементы List можно используя цикл for с индексами, цикл foreach, метод forEach(), итератор или стримы.

List<String> list = new ArrayList<>();  
  
// Using for loop with index  
for (int i = 0; i < list.size(); i++) {  
String element = list.get(i);  
// ….  
}  
  
// Using foreach loop  
 for (String element : list) {  
 // ….  
 }  
  
// Using forEach() method  
 list.forEach(element -> {  
 // ….  
 });  
  
// Using iterator  
Iterator<String> iterator = list.iterator();  
while (iterator.hasNext()) {  
String element = iterator.next();  
// ….  
}  
  
// Using streams  
 list.stream().forEach(element -> {  
 // ….  
 });

1. Какие коллекции синхронизированы?

В Java, многие стандартные коллекции, такие как ArrayList, HashSet, HashMap, LinkedList, TreeSet, TreeMap, Vector и Stack, являются синхронизированными. Это означает, что их методы являются синхронизированными и могут безопасно использоваться в многопоточных приложениях.

1. Какую функциональность представляет класс Collections?

Класс Collections в Java предоставляет статические методы для работы с коллекциями. Он содержит методы для сортировки, перемешивания, поиска, замены элементов и другие операции, которые могут быть полезны при работе с коллекциями.

1. Почему Map не наследуется от Collection?

Map является интерфейсом в Java, который представляет собой объект, который сопоставляет ключи с значениями. Collection, напротив, является интерфейсом, который представляет собой группу объектов (называемых элементами). Map не наследуется от Collection, потому что Map представляет собой структуру данных, которая не является коллекцией элементов, а является структурой, которая связывает ключи с значениями.

1. Почему в классе iterator нет метода для получения следующего элемента без передвижения курсора?

Метод next() в классе Iterator используется для получения следующего элемента и передвигает курсор итератора. Если бы существовал метод для получения следующего элемента без передвижения курсора, то это было бы противоречит логике работы итератора.

1. Почему нет метода iterator.add() чтобы добавить элементы в коллекцию?

Метод add() в классе Iterator предназначен для добавления новых элементов в коллекцию, которая поддерживает итерацию. Однако, не все коллекции поддерживают этот функциональный интерфейс Iterator. Например, ListIterator, который расширяет Iterator, предоставляет дополнительные методы для добавления и удаления элементов, а также для изменения текущего элемента.

1. Расскажите о реализации классов HashSet и TreeSet?

HashSet реализует интерфейс Set и использует хеш-таблицу для хранения элементов. Он не гарантирует никакого порядка итерации.

TreeSet реализует интерфейс NavigableSet и использует красно-черное дерево для хранения элементов. Он гарантирует, что элементы будут отсортированы в соответствии с их значениями или порядком, определенным компаратором.

1. Чем отличаются ArrayDequeи TreeMap? Как они устроены и работают? Что со временем доступа к объектам, какие зависимости?

ArrayDeque реализует интерфейс Deque и использует массив для хранения элементов. Он поддерживает быстрый доступ к обоих концам очереди, но добавление или удаление элементов в середине может быть медленным, так как требует сдвига элементов.

TreeMap реализует интерфейс NavigableMap и использует красно-черное дерево для хранения пар ключ-значение. Он гарантирует, что элементы будут отсортированы по ключам или порядку, определенному компаратором.

1. Что будет, если в Map положить два значения с одинаковым ключом?

Если в Map положить два значения с одинаковым ключом, то старое значение будет заменено на новое. В HashMap и TreeMap это будет происходить независимо от того, какое значение было добавлено первым.

1. Что делать, чтобы не возникло исключение ConcurrentModificationException?

ConcurrentModificationException возникает, когда итерируется по коллекции, которая была изменена после создания итератора. Чтобы избежать этого исключения, вы можете использовать итераторы, которые предоставляются коллекциями, поддерживающими fail-fast поведение, такими как ArrayList и HashMap. Вы также можете использовать коллекции, которые предоставляют fail-safe итераторы, такие как CopyOnWriteArrayList и ConcurrentHashMap.

1. Что такое Hashtable, чем она отличается от HashMap? На сегодняшний день она deprecated, как все-таки использовать нужную функциональность?

Hashtable является устаревшей реализацией Map, которая синхронизирована для потокобезопасности. HashMap - это не синхронизированная версия Hashtable, которая обычно предпочтительнее для многопоточных приложений. Вместо Hashtable вы можете использовать ConcurrentHashMap, который предоставляет более высокую производительность и позволяет безопасный доступ из нескольких потоков.