Day 12 - 25th June 2025

Data Structures

Data structures in Java.docx in docs to study

A ===>             how the character A is stored?

A ====> ASCII = American Standard code for information interchange    value 65 ====> binary code

ASCII values are given ===  by ASNI === American Standard National Institute

Special  before 65

A  ===> 65 …. 66 = B…..

Small characters …  97 = a. 98 = b

A

65 ===> binary code?

65 / 2 == 1

32 / 2 == 0

16 / 2 == 0

8   / 2 == 0

4   / 2 == 0

2   / 2 == 0

1  /     == 1       = ⇒ 1 0 0 0 0 0 1

Bottom to top take the remainders

Task 1:

What is the binary 8 bit representation of A?

ASCII value of 'A' = 65 Binary: 01000001

Task 2:

What is the binary value of a?

Hint ascii value is 97..

ASCII value of 'a' = 97 Binary: 01100001

Task 3: Types of Computer memory with examples.. Explain ..

**Primary Memory (Volatile):**

* **RAM:** Main memory for running programs (DDR4, DDR5)
* **Cache:** High-speed memory in CPU (L1, L2, L3 cache)

**Secondary Memory (Non-volatile):**

* **HDD/SSD:** Permanent storage for files and programs
* **Optical Media:** CDs, DVDs, Blu-ray discs

**Key Difference:** Primary memory is fast but temporary (lost when power off), while secondary memory is slower but permanent storage.

Task 4:

What do you understand by data structures..?

**Data structures are organized ways to store and manage data efficiently in computer memory.** They determine how we can access, insert, delete, and manipulate information based on specific use cases.

**Different structures excel at different operations** - arrays for fast random access, linked lists for dynamic insertion/deletion, stacks for LIFO operations, queues for FIFO processing, trees for hierarchical data, and hash tables for quick lookups.

**Choosing the right data structure directly impacts program performance** - using an array when you need frequent insertions wastes time, while using a linked list for random access is inefficient.

**They form the foundation for algorithms and software design** - understanding data structures helps you write faster, more memory-efficient code and solve complex problems systematically.

Task 5:

What are the operations on data structures ?

**Operations on Data Structures:**

**Insertion:** Adding new elements to the data structure (push, enqueue, insert at position).

**Deletion:** Removing elements from the data structure (pop, dequeue, delete by value/position).

**Traversal:** Visiting all elements in the structure (iterate through array, tree traversal, graph traversal).

**Search:** Finding specific elements or checking if they exist (linear search, binary search, hash lookup).

**Update/Modification:** Changing values of existing elements and sorting/merging operations for reorganizing data.

Task 6:

What are static and dynamic arrays key points summarize in a table

Size, performance, memory, flexibility

| **Aspect** | **Static Arrays** | **Dynamic Arrays** |
| --- | --- | --- |
| **Size** | Fixed at compile/declaration time | Can change during runtime |
| **Memory Location** | Stack (usually) | Heap |
| **Declaration** | int arr[10]; or int[] arr = new int[5]; | ArrayList<Integer> list = new ArrayList<>(); |
| **Memory Allocation** | All memory allocated at once | Memory allocated as needed |
| **Performance** | Faster access (direct indexing) | Slightly slower (overhead for resizing) |
| **Memory Usage** | May waste space if not fully used | Efficient memory usage |
| **Flexibility** | Limited - cannot resize | High - can grow/shrink dynamically |
| **Examples in Java** | int[] arr, String[] names | ArrayList, Vector, LinkedList |

Task 7:

Reverse an array. write a code.

Hint : take a list of nos and display in reverse order..

public class ReverseArray {

public static void main(String[] args) {

int[] arr = {1, 2, 3, 4, 5, 6, 7, 8};

System.out.println("Original Array:");

printArray(arr);

// Method 1: Two-pointer approach (in-place reversal)

reverseArray(arr);

System.out.println("Reversed Array:");

printArray(arr);

}

public static void reverseArray(int[] arr) {

int start = 0;

int end = arr.length - 1;

while (start < end) {

// Swap elements

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

public static void printArray(int[] arr) {

for (int i = 0; i < arr.length; i++) {

System.out.print(arr[i] + " ");

}

System.out.println();

}

}

Task 8:

Reverse a string .. write a code.

Hint: take a name from the user and display the name in reverse order..

import java.util.Scanner;

public class ReverseString {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

// Method 1: Using StringBuilder (most efficient)

String reversed1 = reverseUsingStringBuilder(name);

// Method 2: Using character array

String reversed2 = reverseUsingCharArray(name);

System.out.println("Original name: " + name);

System.out.println("Reversed name (StringBuilder): " + reversed1);

System.out.println("Reversed name (Char Array): " + reversed2);

scanner.close();

}

// Method 1: Using StringBuilder

public static String reverseUsingStringBuilder(String str) {

StringBuilder sb = new StringBuilder(str);

return sb.reverse().toString();

}

// Method 2: Using character array

public static String reverseUsingCharArray(String str) {

char[] charArray = str.toCharArray();

int start = 0;

int end = charArray.length - 1;

while (start < end) {

char temp = charArray[start];

charArray[start] = charArray[end];

charArray[end] = temp;

start++;

end--;

}

return new String(charArray);

}

}

Task 9:

Leetcode and Hackerrank … accounts ..

By Zain – link given for practice

[AlgoMaster.io - Master Software Engineering Interviews](https://algomaster.io/)

Task 10:

public class Example {

   public static void main (String[] args) {

      int[] arr1 = {11, 34, 66, 75};

      int n1 = arr1.length;

      int[] arr2 = {1, 5, 19, 50, 89, 100};

      int n2 = arr2.length;

      int[] merge = new int[n1 + n2];

      int i = 0, j = 0, k = 0, x;

      System.out.print("Array 1: ");

      for (x = 0; x < n1; x++)

      System.out.print(arr1[x] + " ");

      System.out.print("\nArray 2: ");

      for (x = 0; x < n2; x++)

      System.out.print(arr2[x] + " ");

      while (i < n1 && j < n2) {

         if (arr1[i] < arr2[j])

            merge[k++] = arr1[i++];

         else

            merge[k++] = arr2[j++];

      }

      while (i < n1)

      merge[k++] = arr1[i++];

      while (j < n2)

      merge[k++] = arr2[j++];

      System.out.print("\nArray after merging: ");

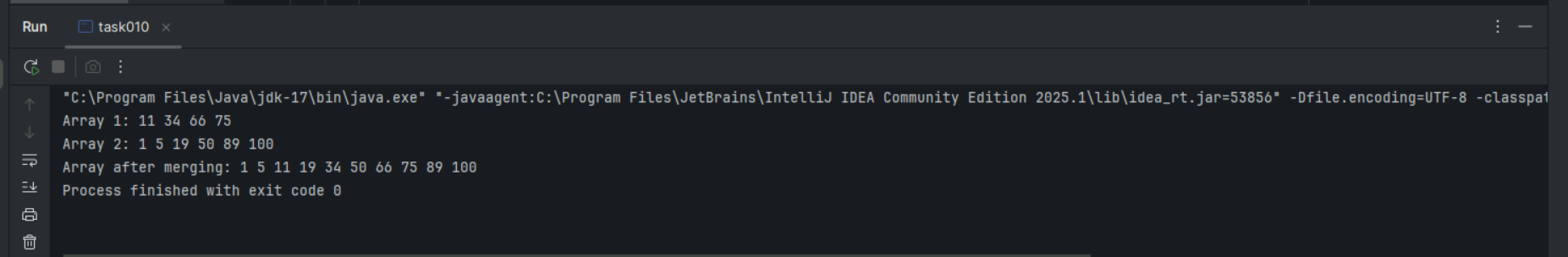
      for (x = 0; x < n1 + n2; x++)

      System.out.print(merge[x] + " ");

   }

}

What is the above code snippet doing..?



Task 11:

What do you know about hash table?

**Hash Table:** A data structure that uses a hash function to map keys to array indices for fast data storage and retrieval.

**How it works:** Hash function converts keys into array positions. Example: key "John" → hash function → index 5 → store value at arr[5].

**Main Operations:** Insert, search, and delete operations typically run in O(1) average time complexity.

**Collision Handling:** When multiple keys hash to same index, resolved using chaining (linked lists) or open addressing (probing for next available slot).

**Use Cases:** Database indexing, caches, dictionaries, and anywhere fast key-value lookups are needed.

Task 12:

Wap to create  a hash table and display them..

Hint 👍

Import java.util.Hashtable;

Import java.util.Map;

import java.util.Hashtable;

import java.util.Map;

public class HashTableDemo {

public static void main(String[] args) {

// Create a Hashtable

Hashtable<String, Integer> studentGrades = new Hashtable<>();

// Insert key-value pairs

studentGrades.put("Alice", 85);

studentGrades.put("Bob", 92);

studentGrades.put("Charlie", 78);

studentGrades.put("Diana", 96);

studentGrades.put("Edward", 88);

System.out.println("Hash Table Contents:");

System.out.println("===================");

// Method 1: Display using enhanced for loop

for (Map.Entry<String, Integer> entry : studentGrades.entrySet()) {

System.out.println("Student: " + entry.getKey() + " | Grade: " + entry.getValue());

}

System.out.println("\nHash Table Size: " + studentGrades.size());

// Demonstrate search operation

String searchStudent = "Bob";

if (studentGrades.containsKey(searchStudent)) {

System.out.println("\n" + searchStudent + "'s grade: " + studentGrades.get(searchStudent));

}

// Display all keys

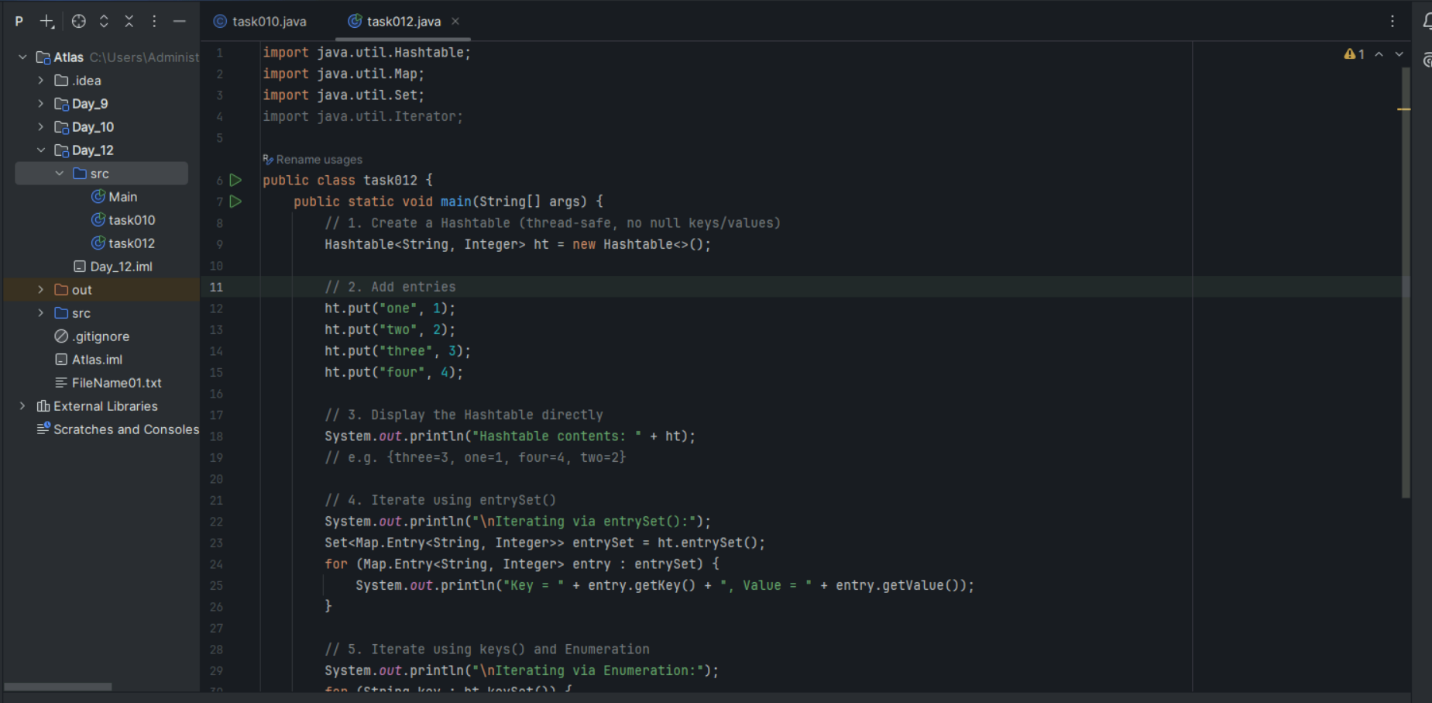
System.out.println("\nAll Students: " + studentGrades.keySet());

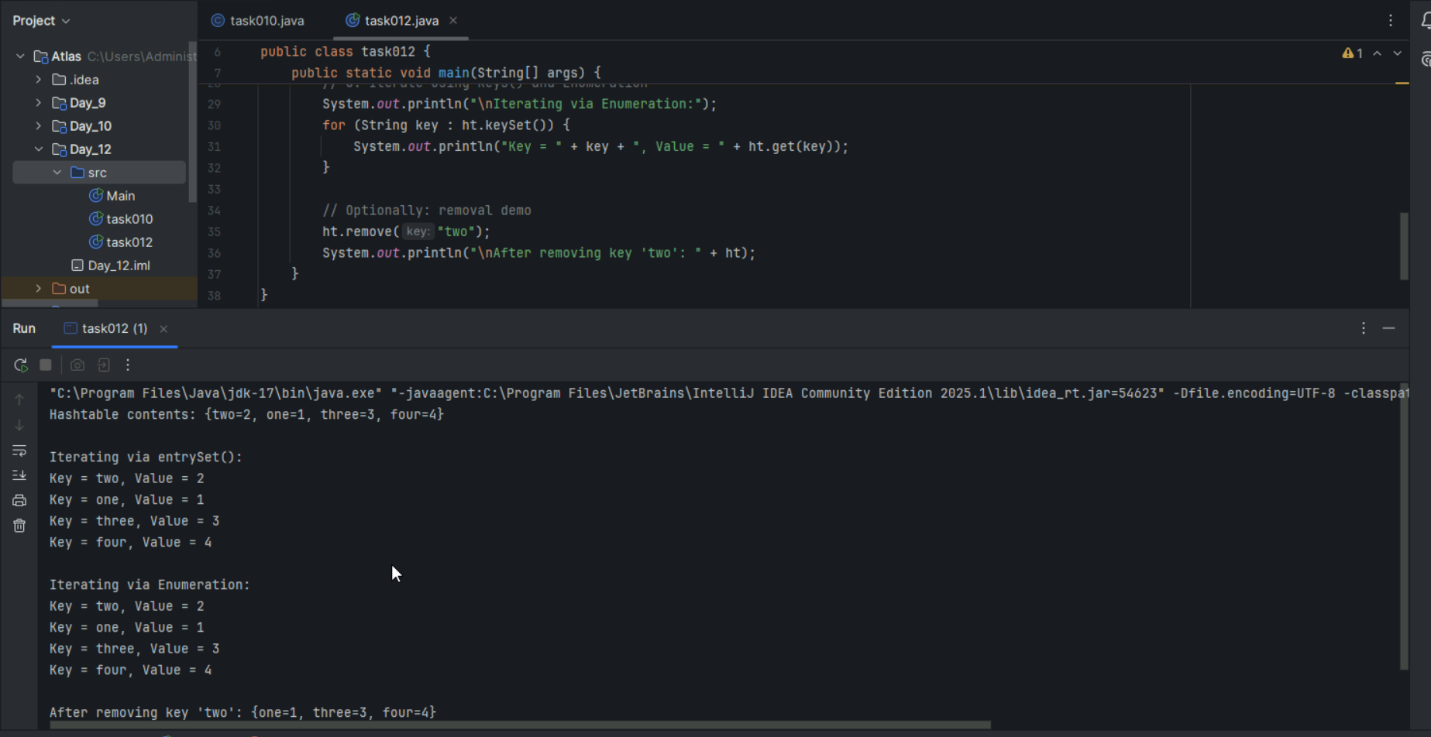
// Display all values

System.out.println("All Grades: " + studentGrades.values());

}

}





Task 13:

import java.util.Hashtable;

import java.util.Map;

public class Task012\_DS\_HashTable {

    public static void main(String[] args) {

        Hashtable<String, Integer> ht = new Hashtable<>();

        ht.put("Anitha", 101);

        ht.put("Kavitha", 102);

        ht.put("Meera", 103);

        for (Map.Entry<String, Integer> e : ht.entrySet())

            System.out.println(e.getKey() + " " + e.getValue());

    }

}

// Map is an interface

// hash table -->   slower , sync , thread safe, no null value accepted

// hash map --> faster while retrieving, asynchro , only one null key and multiple null values..

\* Wap to create  a hash map and display them..

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

public class Task013\_DS\_HashMap {

public static void main(String[] args) {

// Create HashMap

HashMap<String, Integer> hm = new HashMap<>();

// Insert key-value pairs (allows one null key and multiple null values)

hm.put("Anitha", 101);

hm.put("Kavitha", 102);

hm.put("Meera", 103);

hm.put(null, 104); // null key allowed

hm.put("Priya", null); // null value allowed

hm.put("Suma", null); // multiple null values allowed

System.out.println("HashMap Contents:");

System.out.println("==================");

// Display using Map.Entry

for (Map.Entry<String, Integer> e : hm.entrySet()) {

System.out.println(e.getKey() + " " + e.getValue());

}

System.out.println("\nUsing keySet() and get():");

System.out.println("==========================");

// Display using keySet

Set<String> keys = hm.keySet();

for (String key : keys) {

System.out.println(key + " " + hm.get(key));

}

System.out.println("\nHashMap Size: " + hm.size());

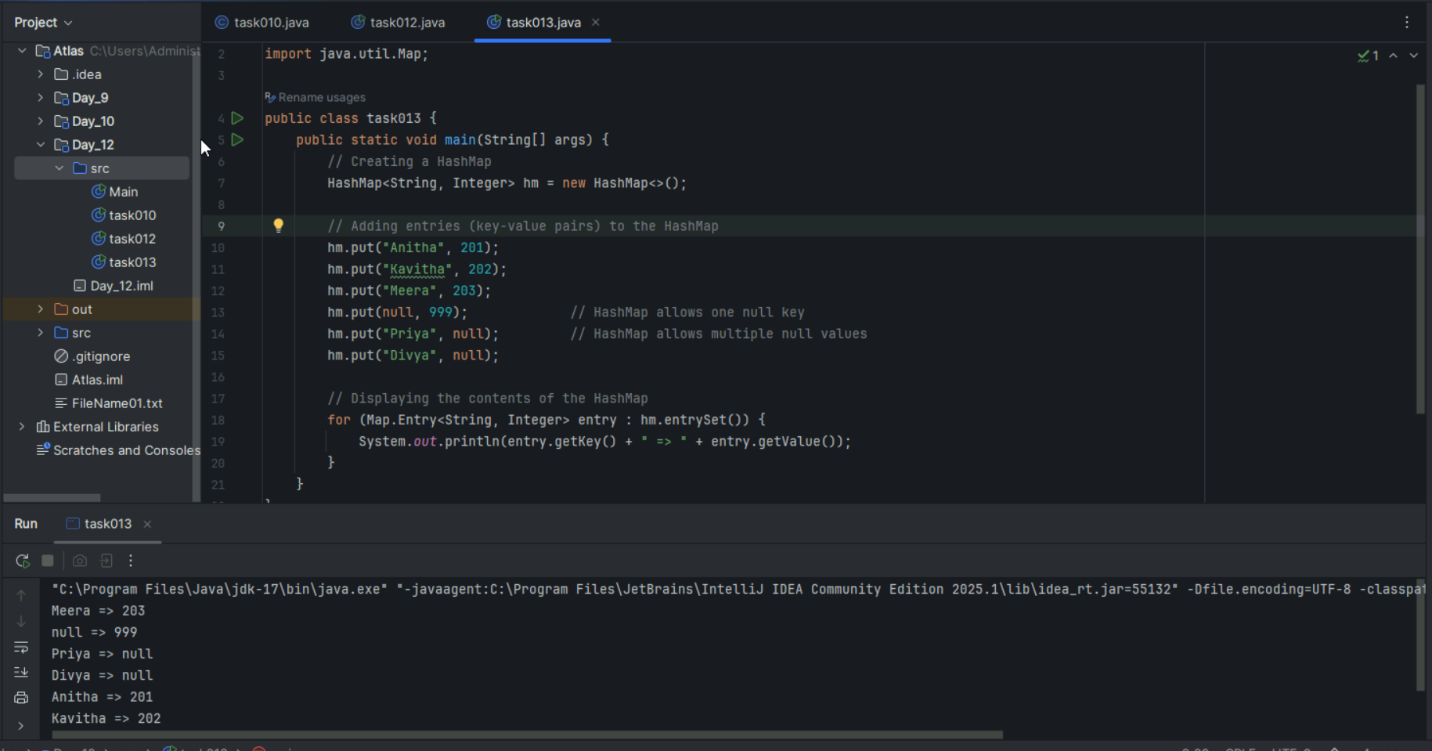
// Demonstrate HashMap features

System.out.println("Contains null key: " + hm.containsKey(null));

System.out.println("Contains null value: " + hm.containsValue(null));

}

}



Task 14:

Hash table advantages and disadvantages

**Hash Table Advantages:**

* **Fast Operations:** O(1) average time for insert, search, and delete operations
* **Efficient Lookup:** Direct access using keys without sequential searching
* **Dynamic Size:** Can grow/shrink during runtime based on data

**Hash Table Disadvantages:**

* **Collision Handling:** Performance degrades when multiple keys hash to same index
* **Memory Overhead:** Extra space needed for handling collisions and maintaining structure
* **No Ordering:** Elements stored randomly, not in any sorted order

Task 15:

Linear probing in Hash table

public class HashTable<Key, Value> {

private class HashTableNode {

private Key key;

private Value value;

private boolean active;

private boolean tombstoned; // Allow reuse of removed slots

public HashTableNode() {

// All nodes in array will begin initialized this way

key = null;

value = null;

active = false;

tombstoned = false;

}

public HashTableNode(Key initKey, Value initData) {

key = initKey;

value = initData;

active = true;

tombstoned = false;

}

}

private final static int TABLE\_SIZE = 9;

private Object[] table;

public HashTable() {

// Since HashNodeTable has generics, we can not have

// a new HashNodeTable[], so use Object[]

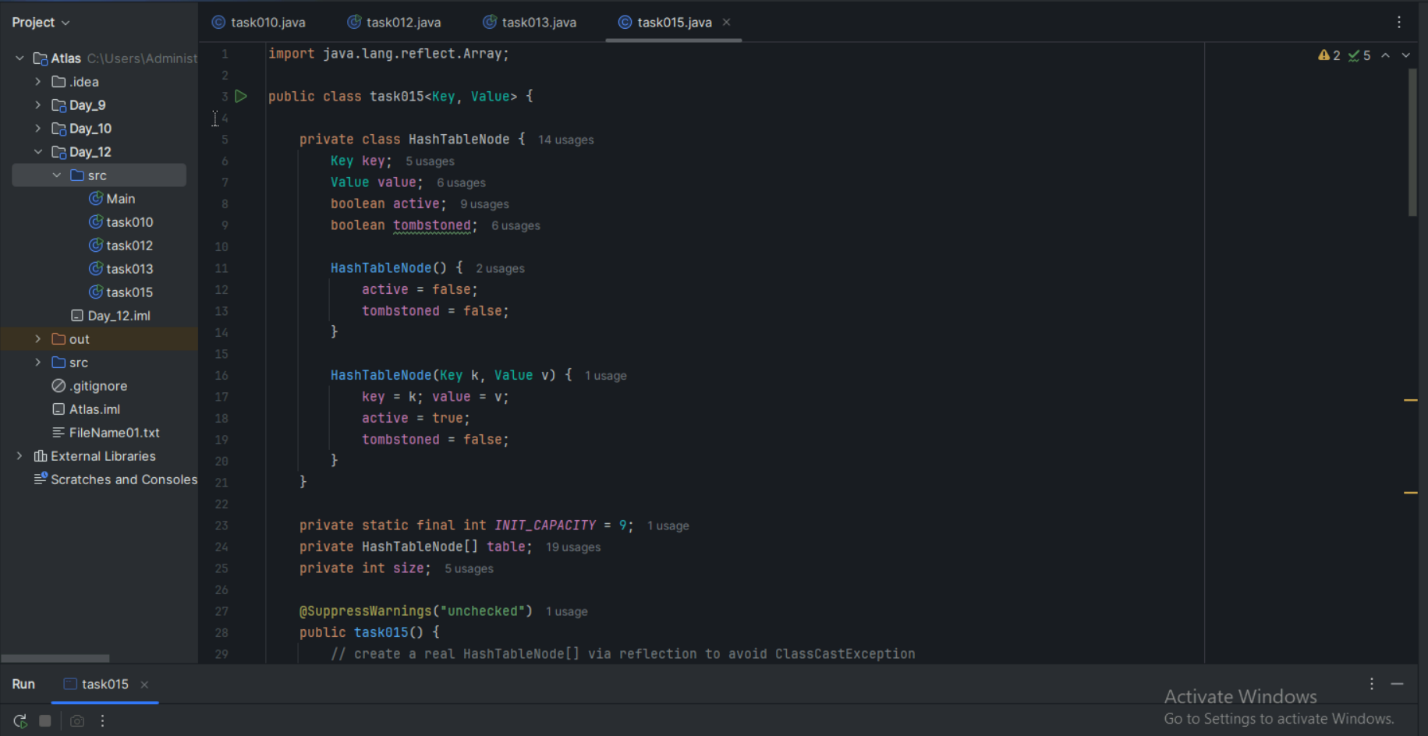
table = new Object[TABLE\_SIZE];

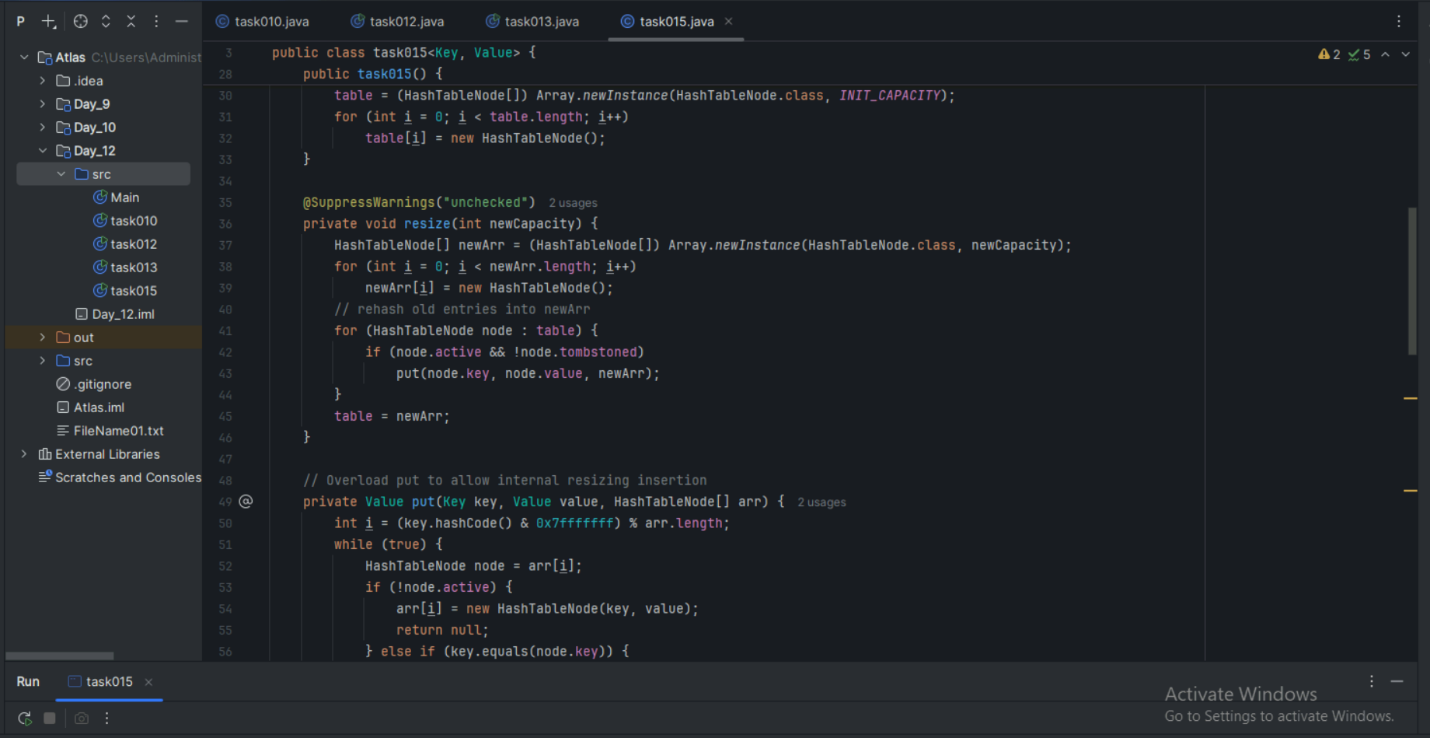
for (int j = 0; j < TABLE\_SIZE; j++)

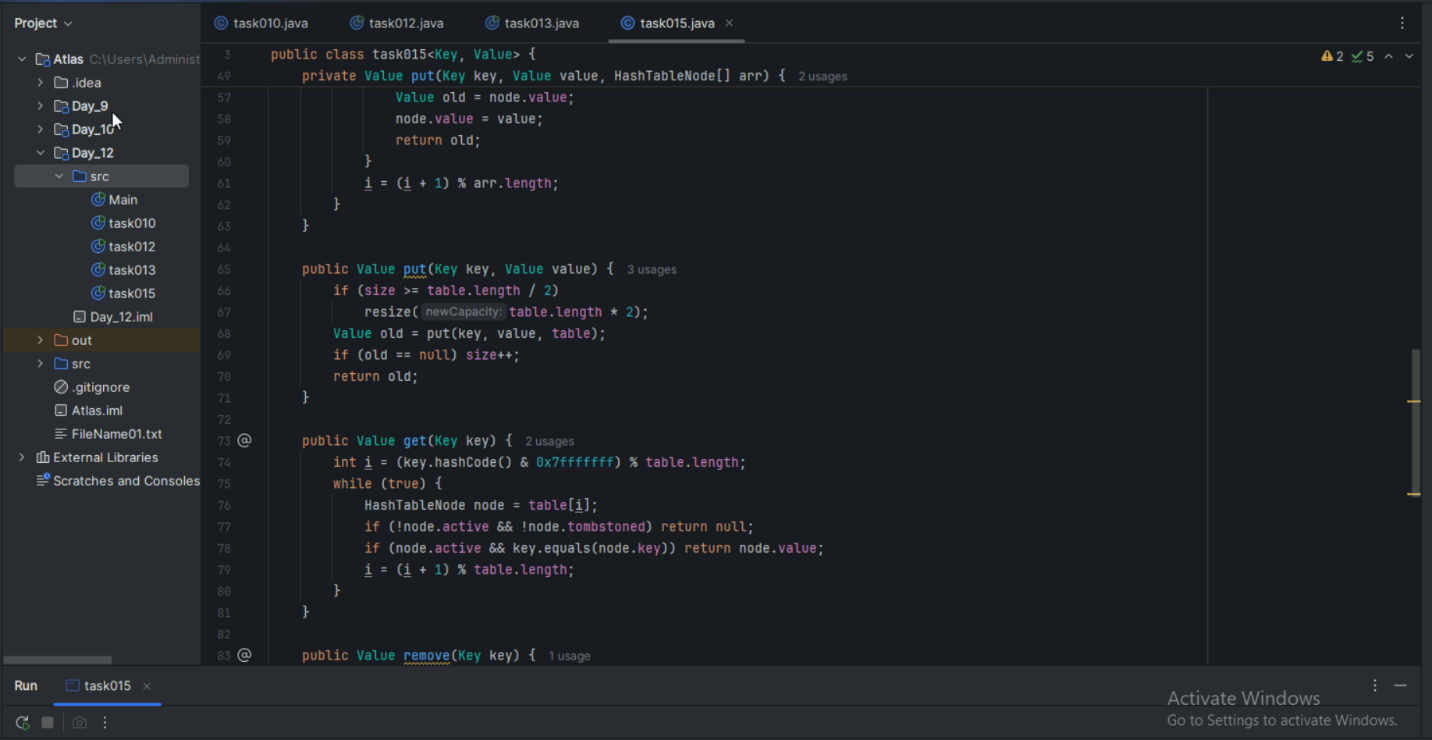
table[j] = new HashTableNode();

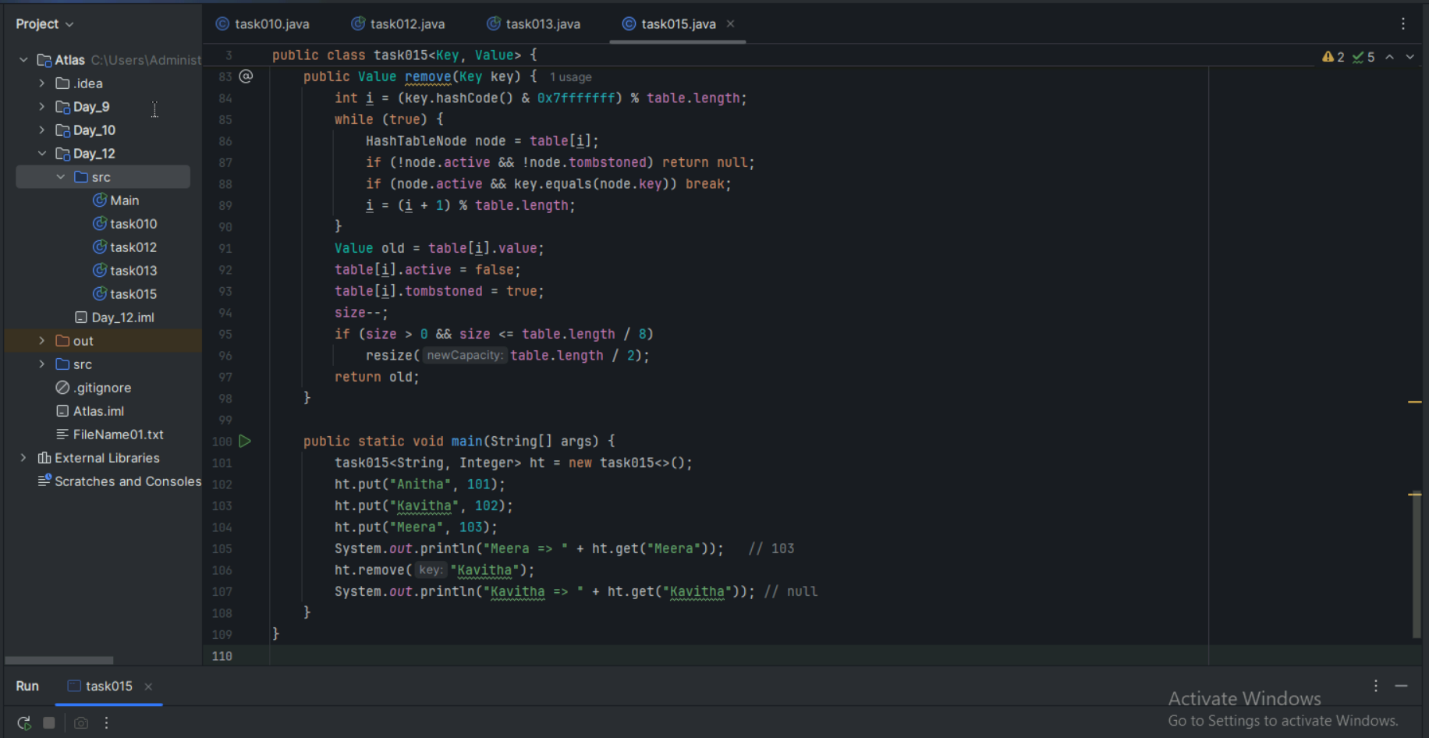
}

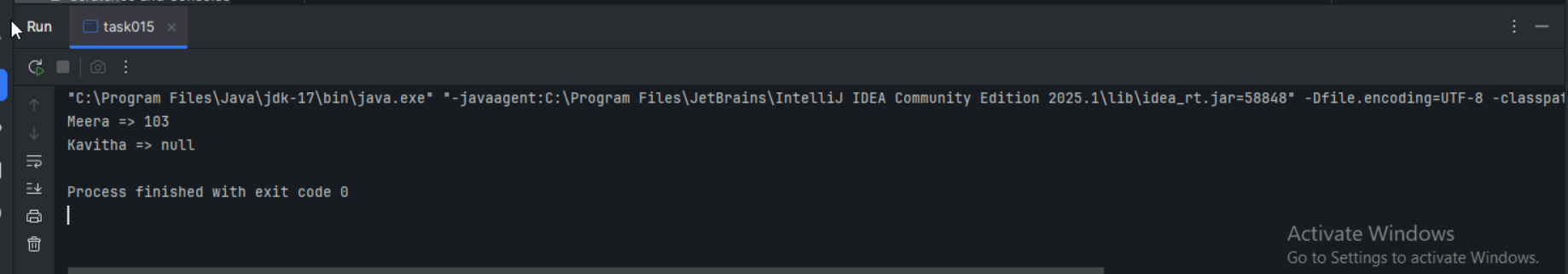
public Value put(Key key, Value value) // TBA











Add on:

Hash Table - Linear Probing

import java.util.Scanner;

class LinearProbingHashTable {

    private int currentSize, maxSize;

    private String[] keys;

    private String[] vals;

    public LinearProbingHashTable(int capacity) {

        currentSize = 0;

        maxSize = capacity;

        keys = new String[maxSize];

        vals = new String[maxSize];

    }

    public void makeEmpty() {

        currentSize = 0;

        keys = new String[maxSize];

        vals = new String[maxSize];

    }

    public int getSize() {

        return currentSize;

    }

    public boolean isFull() {

        return currentSize == maxSize;

    }

    public boolean isEmpty() {

        return getSize() == 0;

    }

    public boolean contains(String key) {

        return get(key) !=  null;

    }

    private int hash(String key)     {

        return key.hashCode() % maxSize;

    }

    public void insert(String key, String val) {

        int tmp = hash(key);

        int i = tmp;

        do {

            if (keys[i] == null) {

                keys[i] = key;

                vals[i] = val;

                currentSize++;

                return;

            }

            if (keys[i].equals(key)) {

                vals[i] = val;

                return;

            }

            i = (i + 1) % maxSize;

        } while (i != tmp);

    }

    public String get(String key) {

        int i = hash(key);

        while (keys[i] != null)

        {

            if (keys[i].equals(key))

                return vals[i];

            i = (i + 1) % maxSize;

        }

        return null;

    }

    public void remove(String key) {

        if (!contains(key))

            return;

        int i = hash(key);

        while (!key.equals(keys[i]))

            i = (i + 1) % maxSize;

        keys[i] = vals[i] = null;

        for (i = (i + 1) % maxSize; keys[i] != null; i = (i + 1) % maxSize) {

            String tmp1 = keys[i], tmp2 = vals[i];

            keys[i] = vals[i] = null;

            currentSize--;

            insert(tmp1, tmp2);

        }

        currentSize--;

    }

    public void printHashTable() {

        System.out.println("\nHash Table: ");

        for (int i = 0; i < maxSize; i++)

            if (keys[i] != null)

                System.out.println(keys[i] +" "+ vals[i]);

        System.out.println();

    }

}

public class LinearProbingHashTableTest {

    public static void main(String[] args)  {

        Scanner scan = new Scanner(System.in);

        System.out.println("Hash Table Test\n\n");

        System.out.println("Enter size");

        LinearProbingHashTable lpht = new LinearProbingHashTable(scan.nextInt() );

        char ch;

        do {

            System.out.println("\nHash Table Operations\n");

            System.out.println("1. insert ");

            System.out.println("2. remove");

            System.out.println("3. get");

            System.out.println("4. clear");

            System.out.println("5. size");

            int choice = scan.nextInt();

            switch (choice) {

            case 1 :

                System.out.println("Enter key and value");

                lpht.insert(scan.next(), scan.next() );

                break;

            case 2 :

                System.out.println("Enter key");

                lpht.remove( scan.next() );

                break;

            case 3 :

                System.out.println("Enter key");

                System.out.println("Value = "+ lpht.get( scan.next() ));

                break;

            case 4 :

                lpht.makeEmpty();

                System.out.println("Hash Table Cleared\n");

                break;

            case 5 :

                System.out.println("Size = "+ lpht.getSize() );

                break;

            default :

                System.out.println("Wrong Entry \n ");

                break;

            }

            lpht.printHashTable();

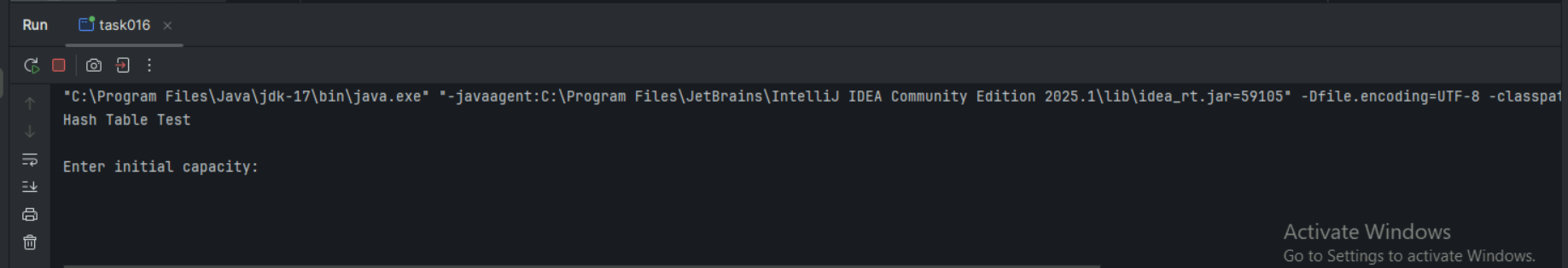
            System.out.println("\nDo you want to continue (Type y or n) \n");

            ch = scan.next().charAt(0);

        } while (ch == 'Y'|| ch == 'y');

    }

}



Task 16:

Methods of Hash table plz list them..   No rating Task

==================================================

Hash table methods List .. for your ref..

* put(K key, V value): Inserts a key-value mapping into the Hashtable. If the key already exists, the old value is replaced with the new one.
* get(Object key): Returns the value associated with the specified key. Returns null if the key is not found.
* remove(Object key): Removes the key-value mapping for the specified key from the Hashtable.
* containsKey(Object key): Returns true if the Hashtable contains a mapping for the specified key, otherwise returns false.
* containsValue(Object value): Returns true if the Hashtable maps one or more keys to the specified value, otherwise returns false.
* isEmpty(): Returns true if the Hashtable contains no key-value mappings, otherwise returns false.
* size(): Returns the number of key-value mappings in the Hashtable.
* clear(): Removes all key-value mappings from the Hashtable.
* keySet(): Returns a Set view of the keys contained in the Hashtable.
* values(): Returns a Collection view of the values contained in the Hashtable.
* entrySet(): Returns a Set view of the key-value mappings contained in the Hashtable.
* rehash(): Increases the size of the Hashtable and rehashes all of its keys. This method is protected and typically handled internally by the Hashtable for performance optimization.
* clone(): Returns a shallow copy of the Hashtable instance.

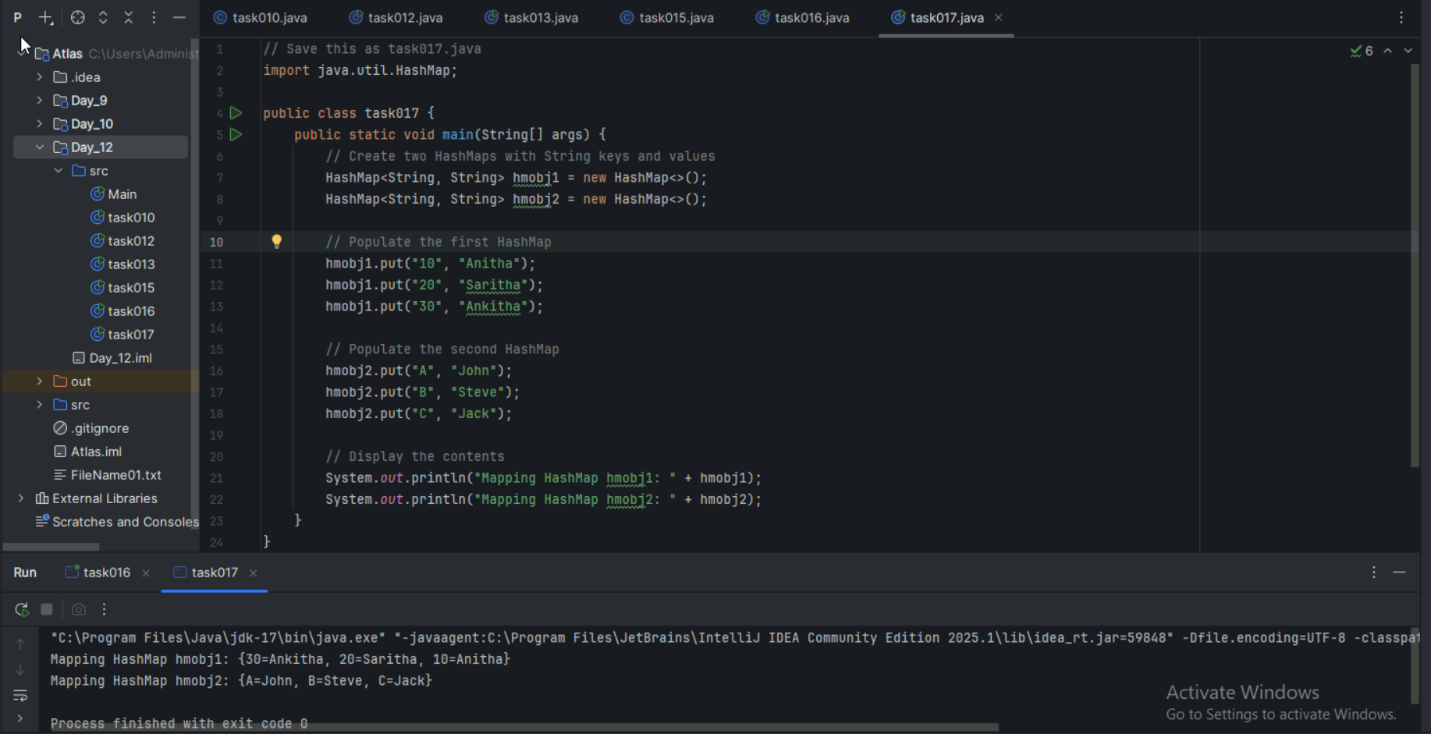
===========================

Task 17:

In Task 13 of hash Map .. we are using string and integer

Can you change to String and string and c if it works

Like this HashMap<String, String>   ? will this work?



Task 18:

Explain the internal working of a HashMap. With diagram.

# HashMap Internal Working

## Core Components

**HashMap**: A data structure that stores key-value pairs using a hash table implementation for fast retrieval.

Uses hash function to convert keys into array indices, allowing direct access to stored values without searching through entire collection.

### 1. ****Bucket Array Structure****

Index: [0] [1] [2] [3] [4] [5] [6] [7] ... [15]

Bucket: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] ... [ ]

### 2. ****Hash Function Process****

Key → hashCode() → Hash Value → Index Calculation

"John" → 2314089 → 2314089 % 16 → Index 9

"Alice" → 63317233 → 63317233 % 16 → Index 1

### 3. ****Node Structure (Entry)****

Node {

int hash; // Hash value of key

K key; // Actual key

V value; // Actual value

Node<K,V> next; // Reference to next node (for chaining)

}

## Internal Working Steps

### ****Step 1: Insertion Process****

1. Calculate hash of key using hashCode()
2. Find bucket index: hash % array\_length
3. Check if bucket is empty:
   * **Empty**: Create new node, place in bucket
   * **Occupied**: Handle collision

### ****Step 2: Collision Handling****

Bucket[1]: Node("Alice", 25) → Node("Bob", 30) → null

(Linked List Chain)

**Before Java 8**: Only Linked List chaining **Java 8+**: Linked List converts to Red-Black Tree when chain length > 8

### ****Step 3: Retrieval Process****

1. Calculate hash of search key
2. Find bucket using hash % array\_length
3. Traverse chain/tree to find exact key match
4. Return associated value

## Visual Representation

HashMap Internal Structure:

==========================

Bucket Array (Default size: 16)

┌─────┬─────┬─────┬─────┬─────┐

│ 0 │ 1 │ 2 │ 3 │ ... │

├─────┼─────┼─────┼─────┼─────┤

│ null│Node │ null│Node │ ... │

└─────┴──┬──┴─────┴──┬──┴─────┘

│ │

▼ ▼

("Alice",25) ("John",30)

│ │

▼ ▼

("Bob",28) null

│

▼

null

## Key Concepts

### ****Load Factor & Resizing****

* **Default Load Factor**: 0.75
* **Threshold**: 16 × 0.75 = 12
* When size > threshold → Array doubles to 32
* All elements are **rehashed** to new positions

### ****Hash Collision Resolution****

1. **Separate Chaining**: Use linked lists/trees
2. **Key Equality**: Uses equals() method for exact match
3. **Tree Conversion**: Chain → Tree when length > 8, Tree → Chain when < 6

### ****Time Complexity****

* **Best Case**: O(1) - Direct bucket access
* **Average Case**: O(1) - Good hash distribution
* **Worst Case**: O(log n) - Tree structure in Java 8+

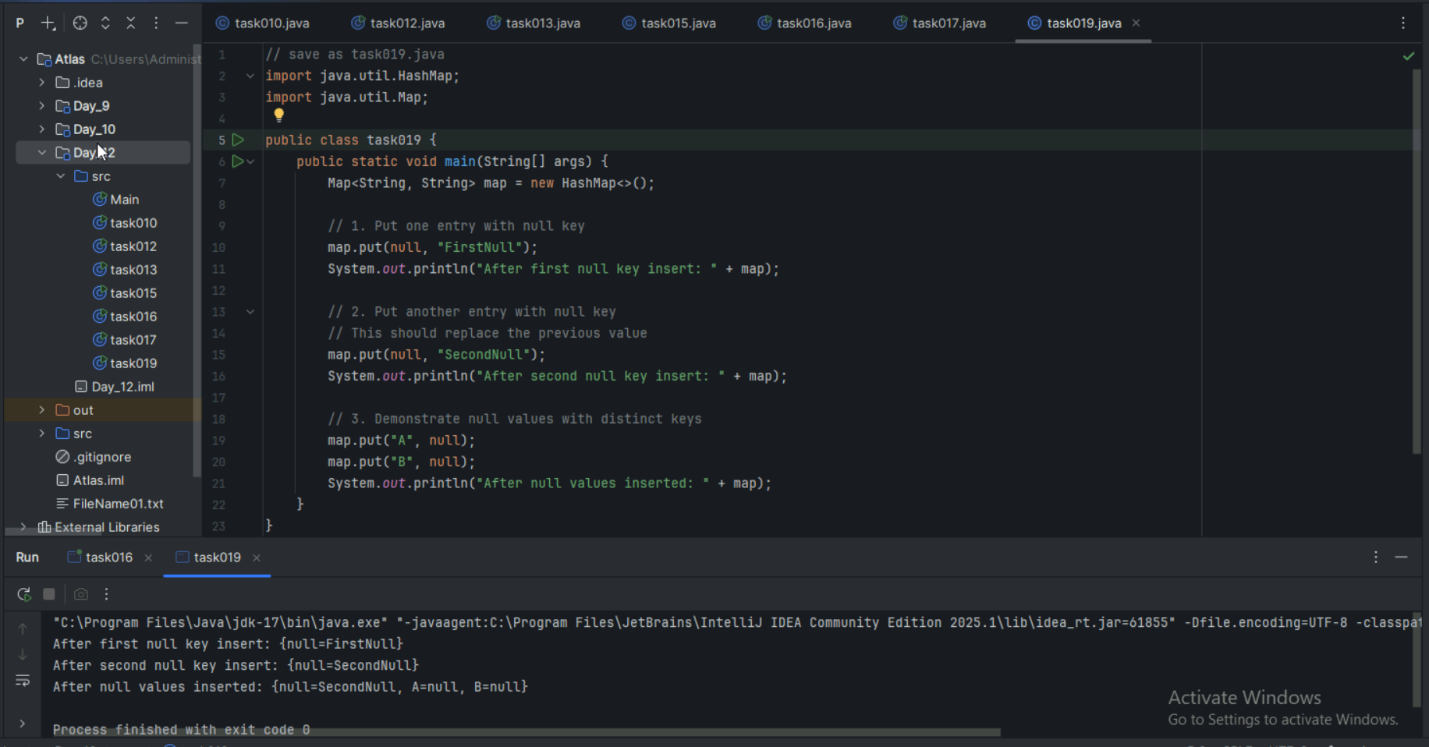
## Important Notes

* Hash function quality affects performance
* Null key stored at index 0
* Bucket index = hash & (n-1) where n is array length (power of 2)
* Rehashing is expensive operation during resizing

Task 19:

Try to add 1 null value in the key and run the hash map code.

Also add one more null value to the key and see the result.



Task 20:

How many methods are there to create a hash Map?

**Methods to Create HashMap:**

Task020\_DS\_HashMapCreateMethods:

Different methods to create a hashmap in java :

1) Constructing a hashmap with default capacity

ex:

 HashMap<String, Integer> hm1 = new HashMap<String, Integer>();

2) Constructing a hashmap with a capacity 10

ex:

HashMap<String, Integer> hm2 = new HashMap<String, Integer>(10);

3)copy one map to another map

ex:

HashMap<String, Integer> hm3 = new HashMap<String, Integer>( hm2);

4)

Specifying load factor along with the capacity

ex:

 HashMap<String, Integer> hm4= new HashMap<String, Integer>(10, 0.75f);

Initial capacity  ===10

Load factor  === 0.75f

Task 21:

Wap to make a Hashmap synchronized..

Plz note : Hash Map are - asynchronous in nature..

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class task021\_DS\_HashMap\_SyncMap {

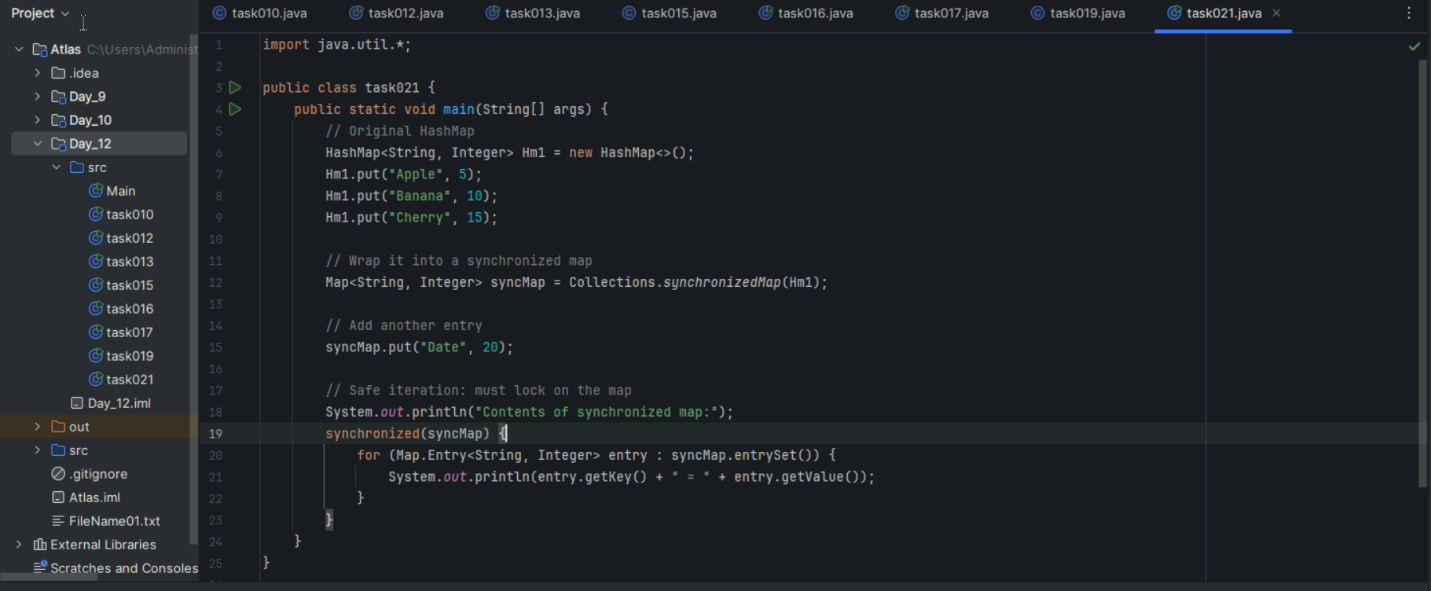
    public static void main(String[] args) {

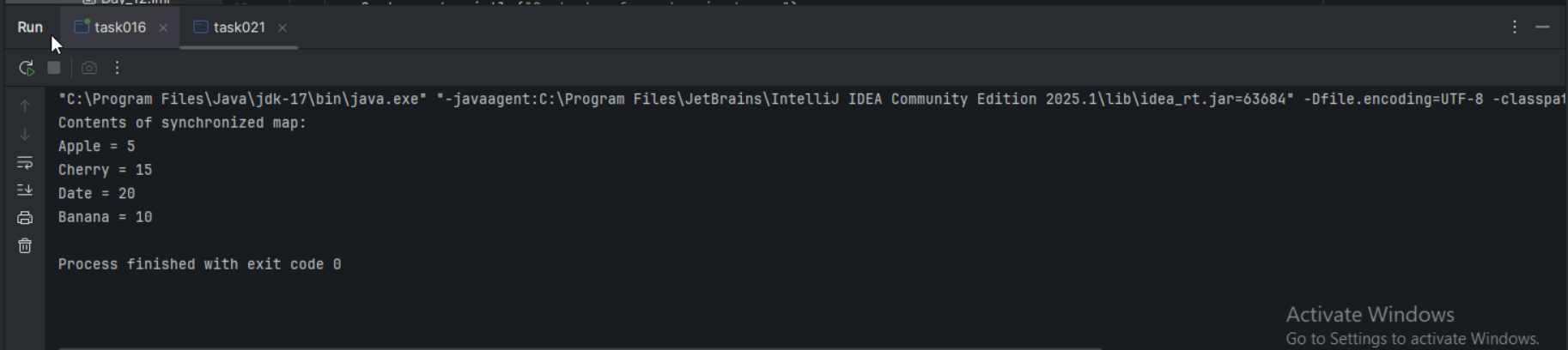
        HashMap<String, Integer> Hm1 = new HashMap<String, Integer>();

        Map<String, Integer> syncMap = Collections.synchronizedMap(Hm1);

   }

}





========================================================================

**Home tasks**:

1. Do hash table have linked list internally?

**Yes, Hash Tables can use linked lists internally for collision resolution.**

**Collision Handling Methods:**

* **Separate Chaining:** Uses linked lists at each bucket to store multiple key-value pairs that hash to same index
* **Open Addressing:** Uses probing techniques without linked lists (linear probing, quadratic probing)

**Java HashMap specifically:** Uses linked lists for chaining. In Java 8+, converts to Red-Black trees when chain length exceeds 8 nodes for better performance.

**Purpose:** Linked lists handle hash collisions when different keys produce the same hash index, allowing multiple entries per bucket.

1. Do collisions occur in hash Maps? What are they?

**Yes, collisions occur frequently in HashMaps.**

**What are Collisions:** When two or more different keys produce the same hash index after hash function calculation.

**Example:** Keys "Alice" and "Bob" both hash to index 5, causing collision at bucket[5].

**Why Collisions Happen:** Hash function maps infinite possible keys to finite array indices. With limited bucket space (default 16), multiple keys inevitably map to same positions.

**HashMap Collision Resolution:** Uses separate chaining - stores collided entries as linked list/tree at same bucket index, maintaining all key-value pairs.