**day12\_107856406\_dsdipt\_sudipto\_25june2025**

**Employee Code:** 107856406

**Login ID:** dsdipt

**Email :** dsdipt@amazon.com

**Name:** Sudipto Das

**Date:** 25 June 2025 (Day 12)

### ***Task 1: What is the binary 8 bit representation of char ‘A’?***

| **Division** | **Quotient** | **Remainder** |
| --- | --- | --- |
| 65 ÷ 2 | 32 | 1 |
| 32 ÷ 2 | 16 | 0 |
| 16 ÷ 2 | 8 | 0 |
| 8 ÷ 2 | 4 | 0 |
| 4 ÷ 2 | 2 | 0 |
| 2 ÷ 2 | 1 | 0 |
| 1 ÷ 2 | 0 | 1 |

Reading remainders **from bottom to top**:

1 0 0 0 0 0 1

To store 'A' as **1 byte (8 bits)**, we add a **leading 0**:

0 1 0 0 0 0 0 1

### ***Task 2: What is the binary 8 bit representation of char ‘a’?***

| **Division** | **Quotient** | **Remainder** |
| --- | --- | --- |
| 97 ÷ 2 | 48 | 1 |
| 48 ÷ 2 | 24 | 0 |
| 24 ÷ 2 | 12 | 0 |
| 12 ÷ 2 | 6 | 0 |
| 6 ÷ 2 | 3 | 0 |
| 3 ÷ 2 | 1 | 1 |
| 1 ÷ 2 | 0 | 1 |

Reading remainders **from bottom to top**:

1 1 0 0 0 0 1

To store 'A' as **1 byte (8 bits)**, we add a **leading 0**:

0 1 1 0 0 0 0 1

### ***Task 3: Explain types of Computer memory with examples.***

Memory refers to **storage areas** where the computer **stores data and instructions** — either **temporarily** or **permanently**.

**🔁 Two Main Types of Memory:**

**1️⃣ Primary Memory (Fast, Temporary)**

Used directly by the **CPU** during processing. It’s **volatile**, meaning it loses data when power is off.

**🔸 A. RAM (Random Access Memory)**

* **Temporary memory** for programs and data in use.
* **Volatile**: Data is lost when the computer shuts down.
* Used for running applications.

**Example:**

* While editing a Word document, it’s in **RAM**.
* 8GB or 16GB RAM in laptops.

⏱ Speed: **Very fast** 💾 Capacity: Limited (typically 4–32 GB)

**🔸 B. ROM (Read-Only Memory)**

* Contains **permanent instructions** like boot code (BIOS).
* **Non-volatile**: Data stays even after shutdown.
* Can’t be changed easily.

**Example:**

* BIOS or UEFI firmware in your computer.
* ROM in calculators, washing machines.

**🔸 C. Cache Memory**

* **High-speed** memory **between CPU and RAM**.
* Stores frequently used data and instructions.
* Located inside or very close to the CPU.

**Example:**

* L1, L2, and L3 cache in processors like Intel i7.

**🔸 D. Registers**

* **Smallest and fastest** memory inside the CPU.
* Stores data **currently being processed**.

**Example:**

* Accumulator register in the CPU holding results of calculations.

**2️⃣ Secondary Memory (Slow, Permanent)**

Used for **long-term storage** of data and programs. It is **non-volatile**.

**🔸 A. Hard Disk Drive (HDD)**

* Stores operating system, files, applications.
* **Magnetic storage**.

**Example:**

* 1TB internal hard drive in a desktop.

**🔸 B. Solid State Drive (SSD)**

* **Faster and more reliable** than HDDs.
* No moving parts (uses flash memory).

**Example:**

* 512GB SSD in a modern laptop.

**🔸 C. Optical Disks**

* Uses **laser technology** to read/write.
* Mostly outdated.

**Example:**

* CD (700MB), DVD (4.7GB)

**🔸 D. Flash Drives / Pen Drives**

* Portable and uses flash memory.
* Plugged into USB ports.

**Example:**

* 16GB or 64GB USB drive.

**🔸 E. Memory Cards**

* Used in **phones, cameras** for external storage.

**Example:**

* 128GB microSD card.

### ***Task 4: What is a Data Structure?***

A **data structure** is a **way of organizing, storing, and managing data** so that it can be **used efficiently**.

**📦 Why Use Data Structures?**

* To **store data efficiently**
* To **retrieve, insert, update, or delete** data quickly
* To solve problems in an **organized and optimized** way

### ***Task 5: Operations on Data Structures***

**🔹 1. Insertion**

* **Add** a new element to the data structure.
* Position can be at the beginning, middle, or end (depending on the structure).

**Examples:**

* Add an element to an array.
* Push a value onto a stack.
* Enqueue in a queue.
* Insert a node in a linked list.

**🔹 2. Deletion**

* **Remove** an element from the data structure.

**Examples:**

* Remove an element from an array.
* Pop from a stack.
* Dequeue from a queue.
* Delete a node in a tree.

**🔹 3. Traversal**

* **Visit** each element of the data structure, usually to display or process it.

**Examples:**

* Loop through an array.
* Traverse a linked list.
* In-order, pre-order, post-order traversal in trees.
* BFS and DFS in graphs.

**🔹 4. Searching**

* **Find** whether an element exists and where it is located.

**Examples:**

* Linear search in arrays.
* Binary search in sorted arrays.
* Search key in a hashmap.
* Search node in a binary search tree.

**🔹 5. Sorting**

* Arrange elements in a **specific order** (ascending or descending).

**Examples:**

* Bubble Sort, Merge Sort, Quick Sort.
* Sorting student marks or names.

**🔹 6. Updating**

* **Change** the value of an existing element.

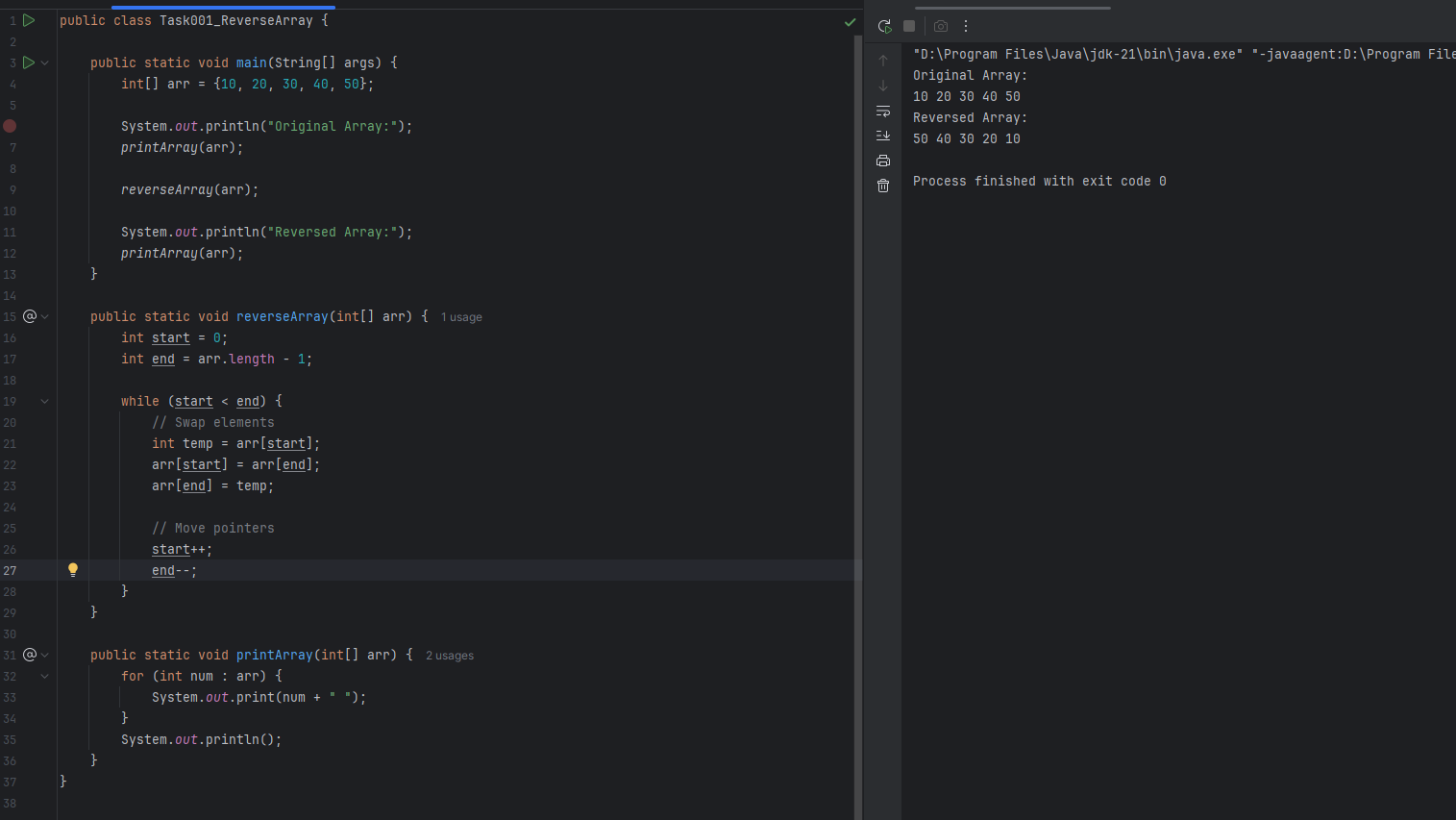
**Examples:**

* Modify the value at a specific index in an array.
* Update a node value in a linked list or tree.

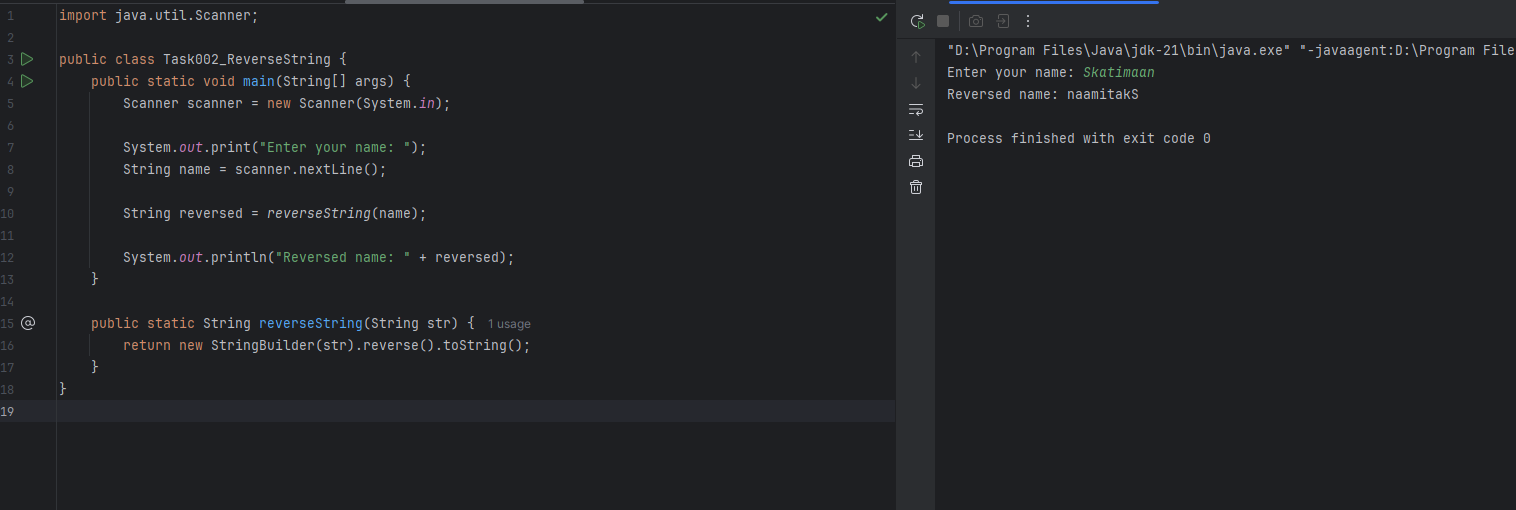
***Task 6: Static vs Dynamic Arrays – Key Points***

| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | Fixed at the time of declaration | Can grow or shrink at runtime |
| **Memory Allocation** | Compile-time | Runtime |
| **Flexibility** | Less flexible | More flexible |
| **Storage** | Contiguous memory | Contiguous memory |
| **Resize Option** | ❌ Not resizable | ✅ Resizable (manually or automatically) |
| **Performance** | Faster due to fixed size | Slightly slower due to resizing overhead |
| **Example (Java)** | int[] arr = new int[5]; | ArrayList<Integer> list = new ArrayList<>(); |
| **Library Support** | Built-in primitive type | Comes from java.util.ArrayList, etc. |
| **Use Case** | When size is known in advance | When size can change during execution |

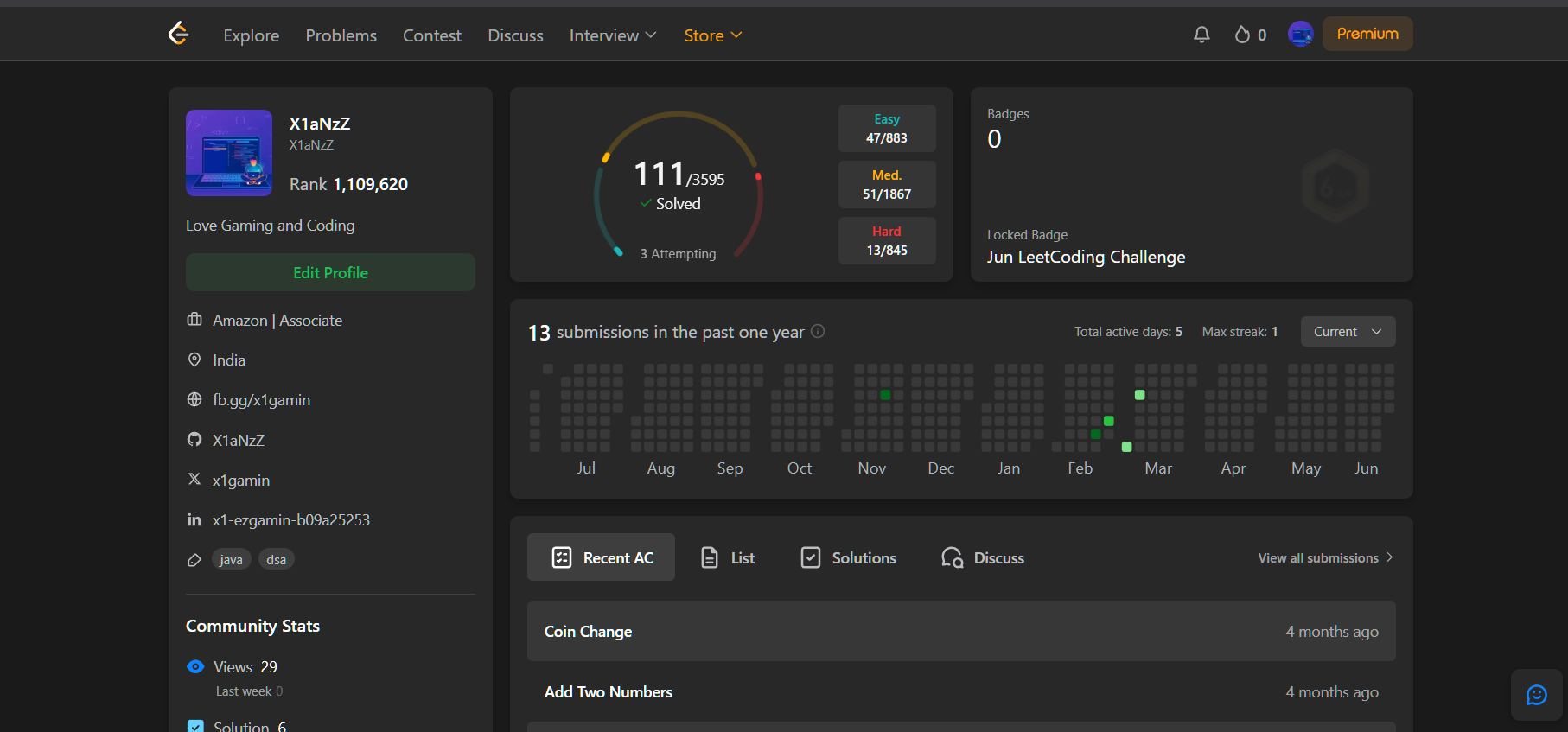
***Task 7: Reverse the elements of an array***

******

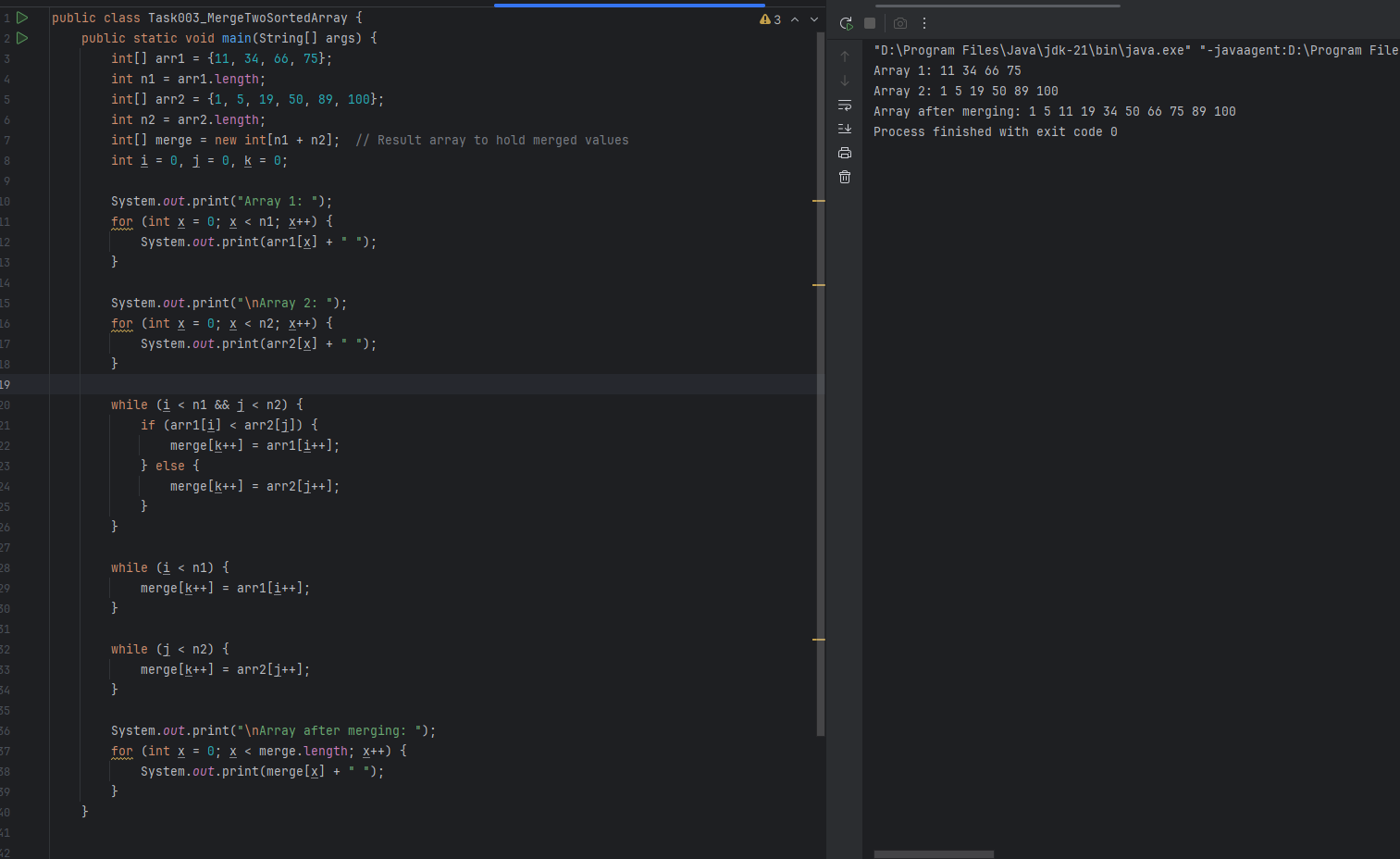
***Task 8: Take user input & reverse the characters in the string***

******

***Task 9: Create LeetCode Account***

******

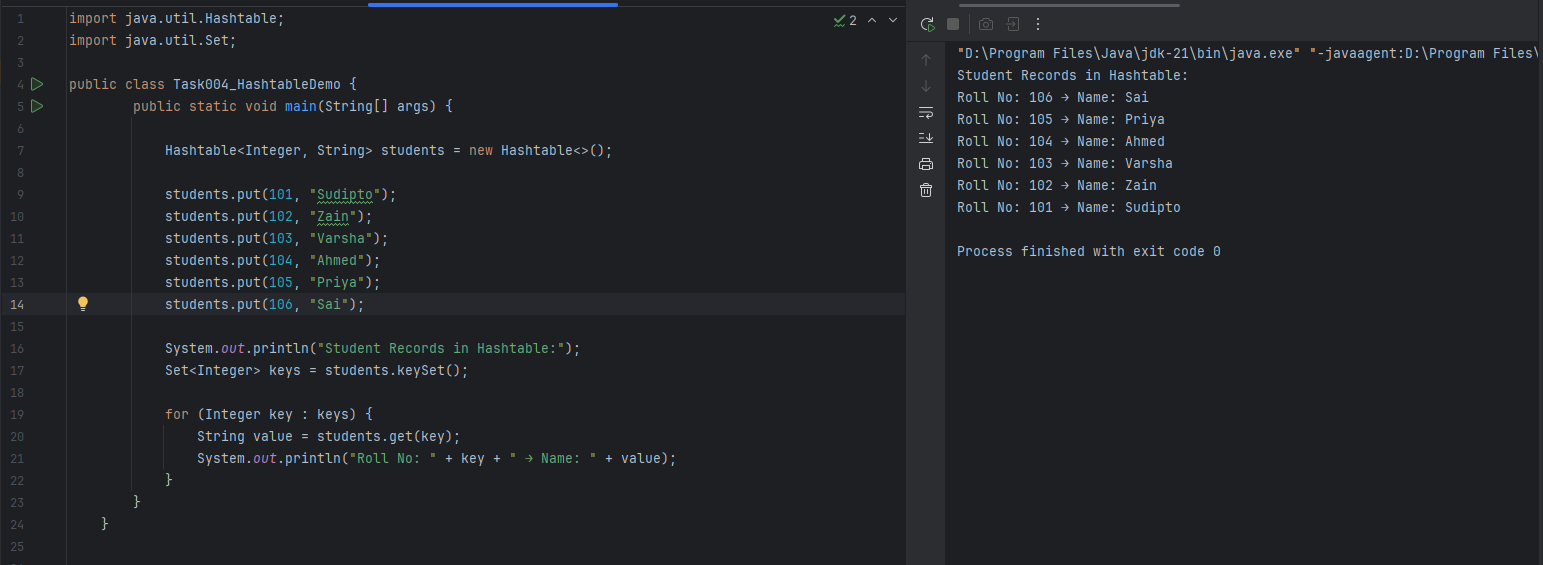
***Task 10: Program to Merge Two Sorted Arrays***

******

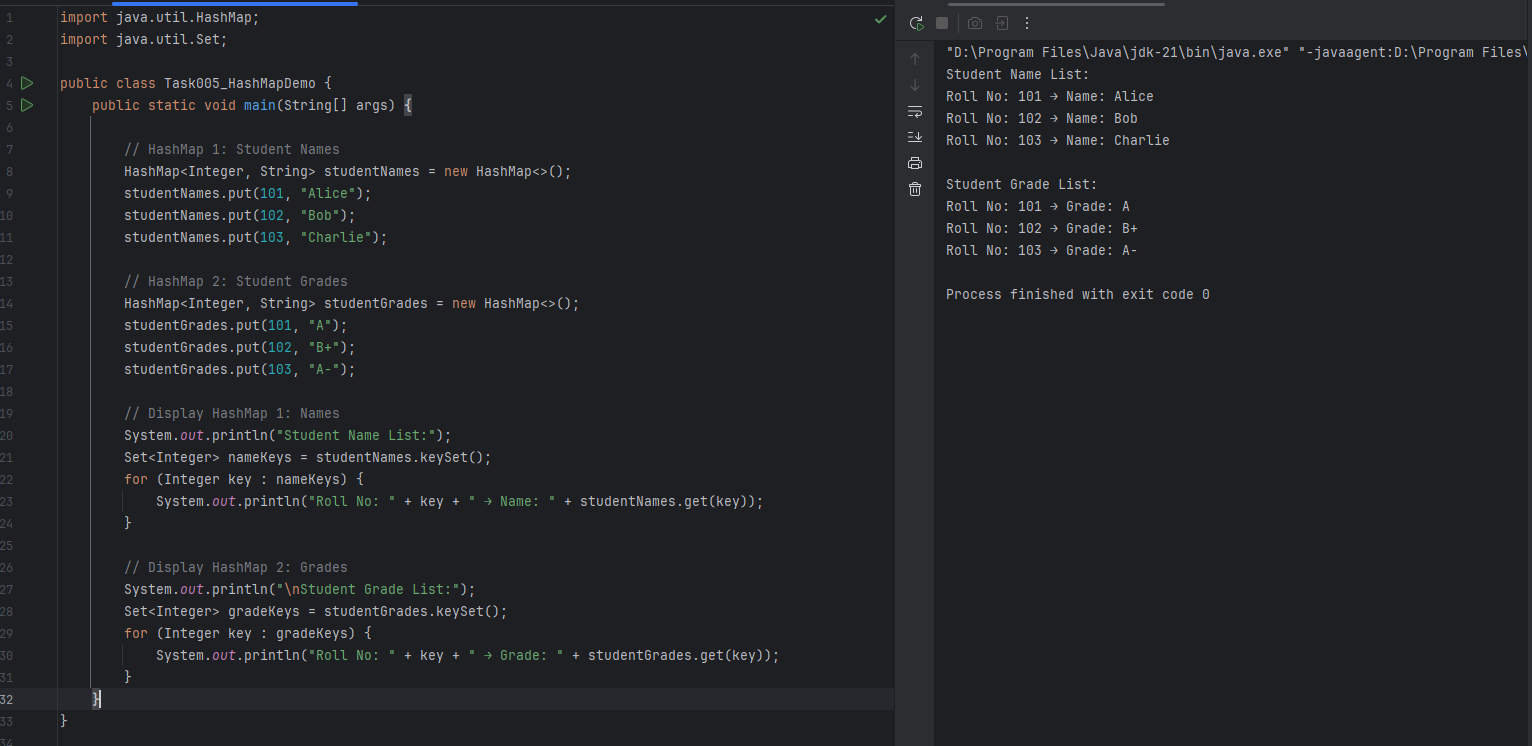
***Task 11: What is a Hash Table?***

A Hash Table is a data structure that stores data in a key-value pair format, and uses a hashing function to compute an index (called a hash code) into an array, where the value will be stored.

***Task 12: Create and Display a Hashtable***



***Task 13: Create and Display a Hashmap***



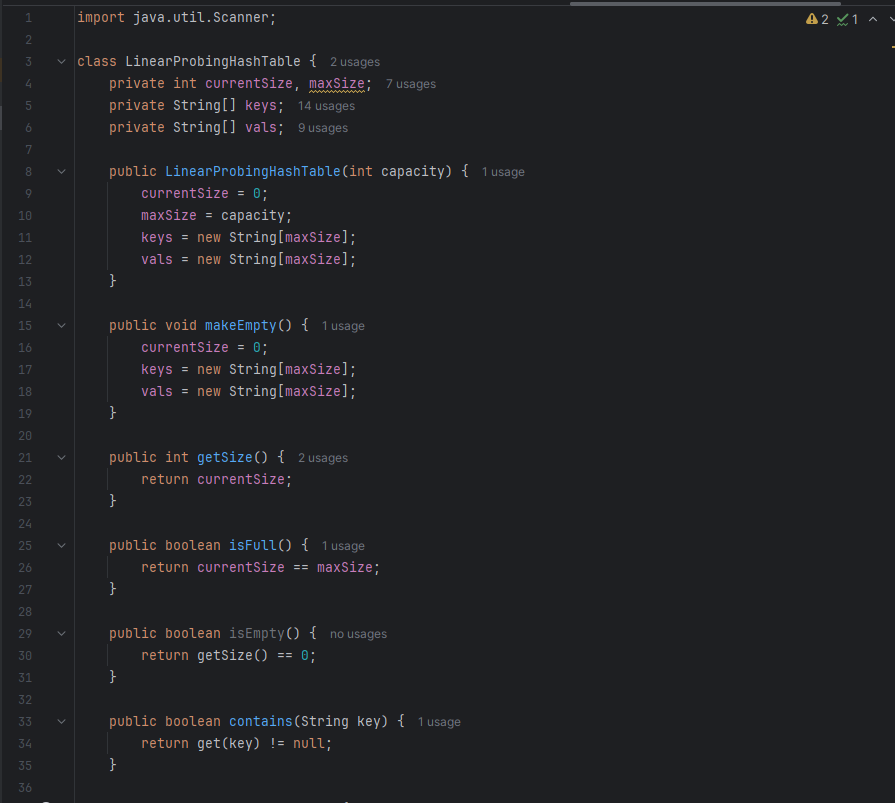
***Task 14: Hash table advantages and disadvantages***

**✅ Advantages of Hash Table**

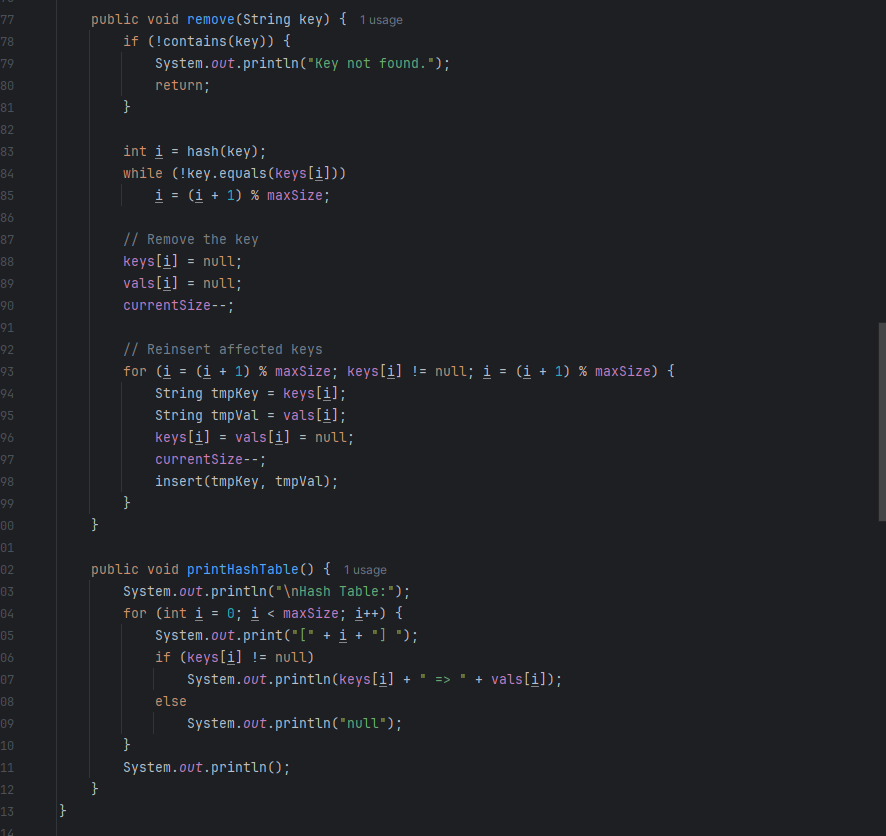
* **Fast data access**: Hash tables offer very fast lookup, insertion, and deletion operations with an average-case time complexity of **O(1)**, making them ideal for high-performance applications.
* **Efficient key-based indexing**: Unlike arrays or lists, you don't need to know the position/index — you can retrieve values directly using unique keys.
* **Scales well with large data**: Hash tables work efficiently even with a large number of records, making them useful in databases, compilers, and caching systems.
* **Prevents duplicate keys**: Each key in a hash table must be unique, which automatically prevents duplication and accidental overwrites.
* **Supports flexible key types**: Most implementations allow different types of keys such as integers, strings, or even custom objects.
* **Widely used in real-world systems**: Hash tables are the underlying structure for HashMap, Dictionary, and symbol tables used in compilers and interpreters.

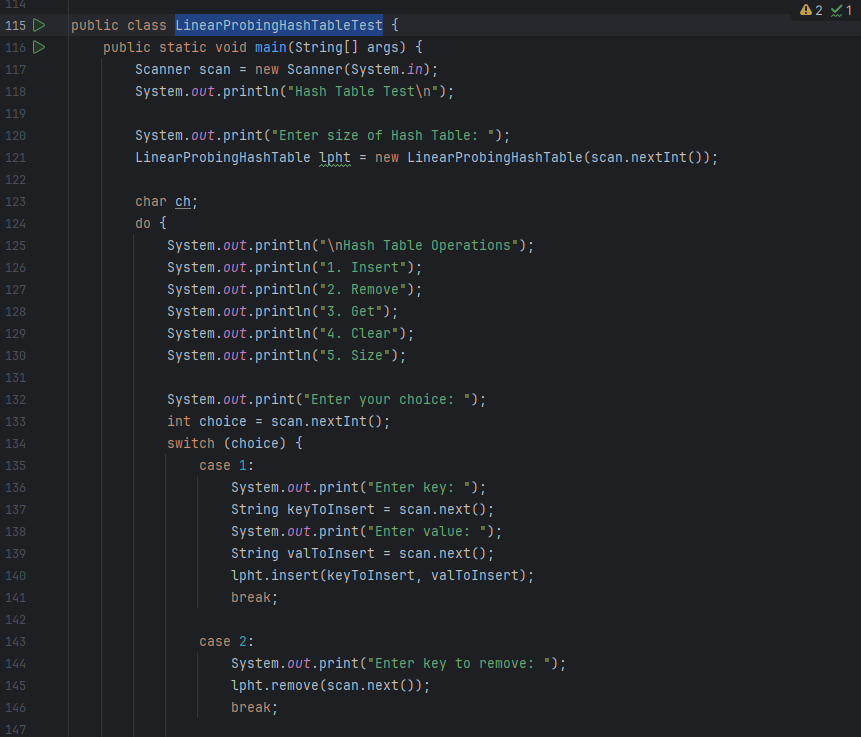
**❌ Disadvantages of Hash Table**

* **Hash collisions can occur**: Two different keys may generate the same hash index, leading to a collision, which requires special handling like chaining or open addressing.
* **No guaranteed order**: Unlike arrays or TreeMaps, hash tables do not maintain any order of keys or values. The data appears in random order.
* **Worst-case performance can be O(n)**: If many collisions happen due to a poor hash function, performance degrades to linear time.
* **Duplicate keys are not allowed**: You can store only one value per key; inserting a new value with an existing key will overwrite the previous value.
* **Requires a good hash function**: The efficiency of the hash table depends on how well the hash function distributes keys.
* **Thread safety issues in some implementations**: For example, Java’s HashMap is not thread-safe and must be handled with synchronization or replaced with ConcurrentHashMap in multi-threaded environments.

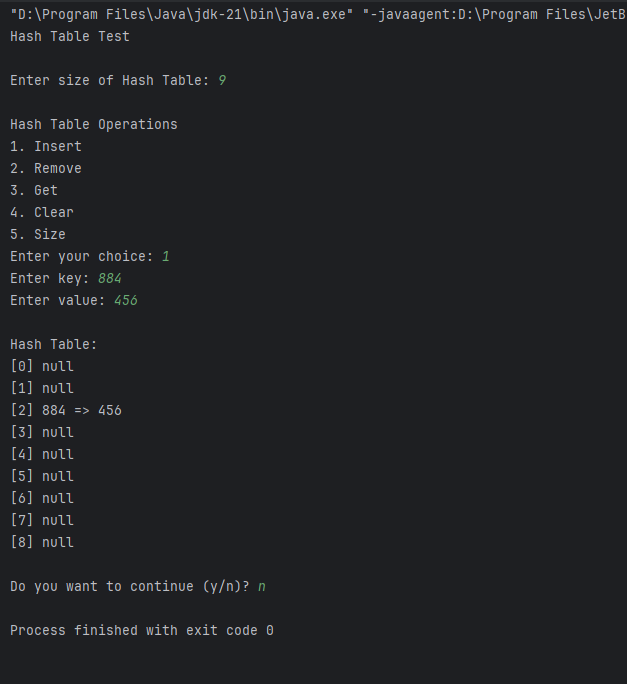
***Task 15: Linear probing in Hash table***   








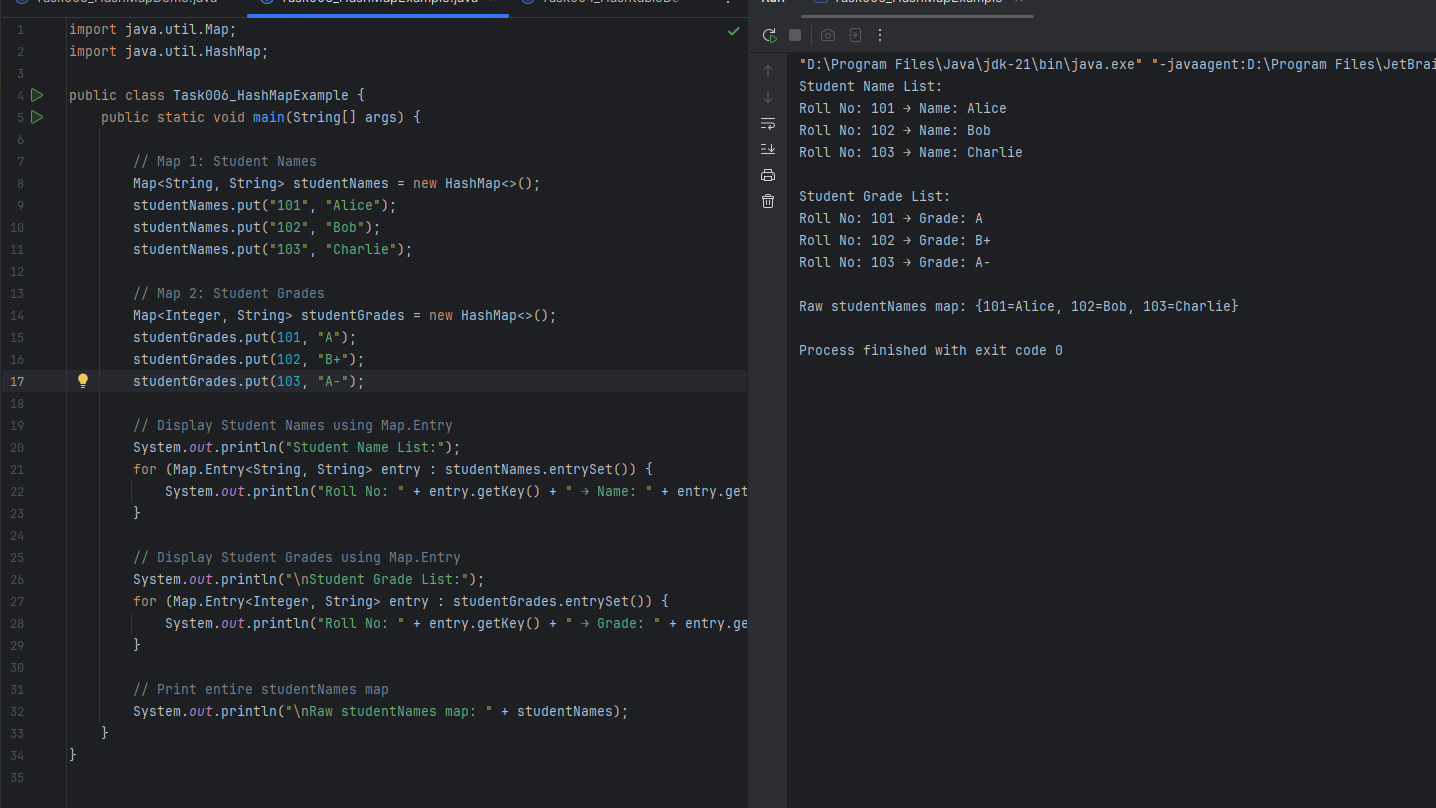




***Task 16: Hash Table Methods List***

| **Method** | **Description** |
| --- | --- |
| put(K key, V value) | Inserts a key-value pair. If the key exists, update the value. |
| get(Object key) | Retrieves the value for the given key. Returns null if not found. |
| remove(Object key) | Deletes the entry for the given key. |
| containsKey(Object key) | Checks if the Hashtable contains the specified key. |
| containsValue(Object value) | Checks if the Hashtable contains the specified value. |
| isEmpty() | Returns true if the Hashtable has no entries. |
| size() | Returns the total number of key-value pairs. |
| clear() | Removes all entries from the Hashtable. |
| keySet() | Returns a Set of all keys. |
| values() | Returns a Collection of all values. |
| entrySet() | Returns a Set of all key-value pairs (Map.Entry). |
| rehash() | Resizes and rehashes the table. It is protected and used internally. |
| clone() | Creates a shallow copy of the Hashtable object. |

***Task 17: Create and Display another Hashmap using <String, String>***



***Task 18: Explain the internal working of a HashMap. With a diagram.***

A HashMap is a part of Java’s **Collection Framework** that stores data in **key-value pairs**. It allows **constant time complexity O(1)** for put and get operations in the **average case**.

**Internal Working of HashMap:**

Suppose we are calling,

map.put("apple", 10);

Java performs the following steps internally:

**Step-by-Step Execution:**

**1. HashCode Calculation:**

* HashMap calls the hashCode() method on the key.

Example:  
int hash = key.hashCode();

**2. Index Calculation:**

* Index in the underlying **array (bucket)** is calculated:  
  index = hash % capacity;
* A **bitwise operation** is used in real implementation for performance:  
  index = (n - 1) & hash;

**3. Bucket Location:**

* Array is used to store nodes.
* If the bucket (index) is empty, the key-value pair is stored directly.
* If not, **collision resolution** is applied.

**4. Collision Resolution:**

* If two keys hash to the same index:
  + Java 7: uses **LinkedList** to store multiple entries.
  + Java 8+: uses **LinkedList**, and if list size > 8, converts to **Red-Black Tree** for performance.

**5. Equality Check:**

* If a key already exists, it uses .equals() to compare and replace the value.

**Internal Class**

Each entry is stored as a **Node**:

static class Node<K,V> implements Map.Entry<K,V> {

final int hash;

final K key;

V value;

Node<K,V> next;

}

**Diagram: HashMap Internal Structure**

Underlying Array (Buckets)

Index | Data

------|-----------------------

0 | null

1 | null

2 | [Key1, Value1] -> [Key2, Value2] // LinkedList (collision)

3 | null

4 | [Key3, Value3]

5 | null

... | ...

**Example:**

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

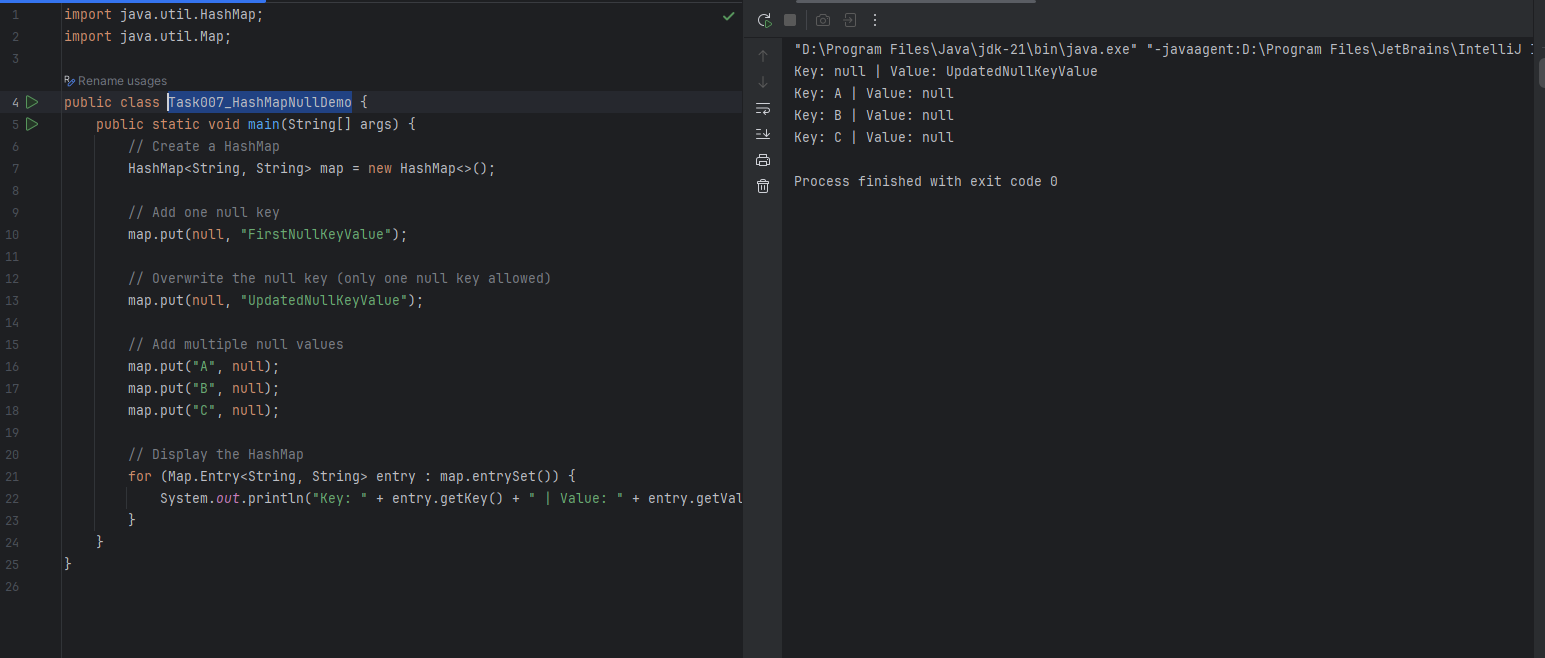
map.put("apple", 10);

map.put("mango", 20);

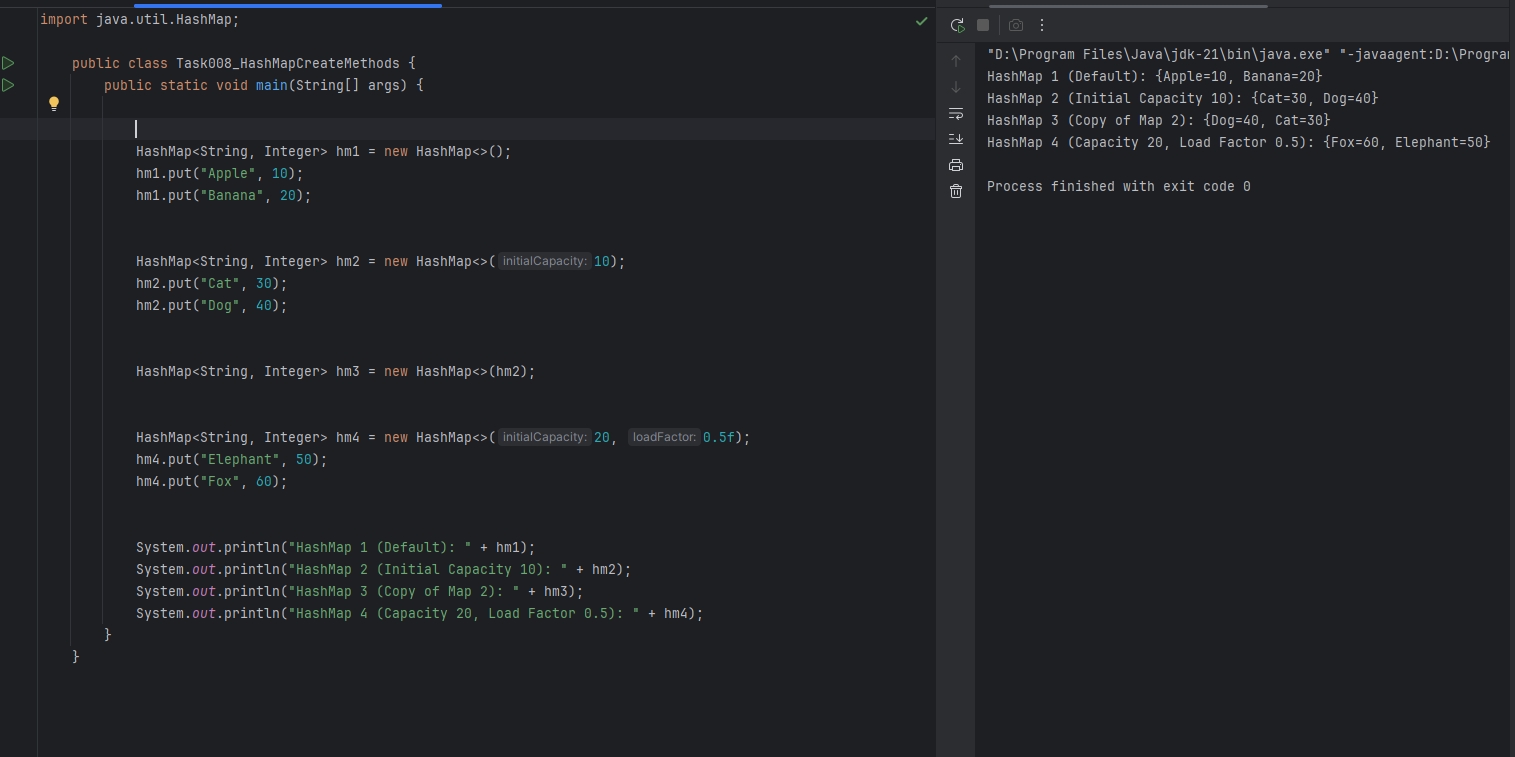
map.put("orange", 30);

If hash("apple") % 16 = 2  
 and hash("mango") % 16 = 2,  
 then both are stored in the **same bucket** → handled by **chaining**.

***Task 19: add one null in key and multiple null in value and display in Hashmap***



***Task 20: Different ways to create a hashmap in java***

******

***Task 21: Make a Hashmap synchronized***

