Task 1:

What do you understand about data structures?

What are Data Structures and why do we use them?

**Data Structures** are like **containers** that help you store, organize, and manage data so that you can use it **efficiently**.

Imagine you're organizing your things at home.

* You put books on a **shelf** → like an **array**
* You keep daily used things in a **queue** to access one after another → like a **queue**
* You keep important contacts in a **phonebook** where you search by name → like a **hashmap**
* You store your clothes in **drawers**, where you only take out the last folded shirt → like a **stack**
* You use a **family tree** to store family members → like a **tree structure**

Each of these is a **data structure**. It’s not just about storing things, but **how** you store them to make life easier.

**Simply, we need Data Structures: If we want to maintain large volumes of data, we need Data Structures.**

**Types of Data Structures:**

Here are some common ones:

| **Type** | **Example in Real Life** | **Use** |
| --- | --- | --- |
| **Array** | List of groceries | Store elements in order |
| **Linked List** | Train coaches | Easy to add/remove items |
| **Stack** | Plates in a stack | LIFO – Last In, First Out |
| **Queue** | People in a line | FIFO – First In, First Out |
| **HashMap / Hash Table** | Dictionary | Quick search by key |
| **Tree** | Family tree | Hierarchical data |
| **Graph** | City map or network | Relationships between things |
|  |  |  |

Arrange the data in such a way that it won’t take time to fetch the data. I.e; the time that is taking for running an app should be in milliseconds.

**Data Structures: A data structure is a smart way of organizing and storing data so that it can be used efficiently in a program.**

Task 2:

What are the types of data structures you know .. list them out..

**Linear Data Structures:** Elements arranged in a Sequential order. i.e; where each element is connected to the one before and after it

Eg: Arrays, Linked List, Stack, Queue

Arrays – List of things. A collection of elements stored in contiguous memory locations with fixed size.

Eg: Like a box with slots to store things in order.

Linked List - A sequence of nodes where each node points to the next one.

Eg: Like a chain of boxes connected together.

Stack - A collection where the last item added is the first one removed (LIFO).

Eg: Like a stack of plates — last in, first out.

Queue - A collection where the first item added is the first one removed (FIFO).

Eg: Like a line of people — first come, first served.

**Non-Linear Data Structures:** Elements not arranged in a sequential order. A **non-linear data structure** arranges data in a hierarchical or interconnected format, where elements are not in a straight line.

Eg: Trees, Graphs.

Trees -A structure with nodes in a hierarchy, starting from a root and branching out.

Eg: Like a family tree — starts from the root and splits into branches.

Graphs -A set of nodes (vertices) connected by edges, used to show relationships.

Eg: Like a city map — places connected by roads.

Task 3:

What all operations can we do in Data structures?

Insertion, Deletion, Searching, sorting, updating, merging, Traversal,

**American National Institute – gave rules like ASCII - ‘A’ – 65…**

When you give the number **5**, the computer doesn't see a "5" like we do — it **converts it** into binary:

Binary code – Low level Language.

What we write / conveying – High level language.

Convert 65 to binary code – 65 /2 = 32

65 ÷ 2 = 32 remainder 1

32 ÷ 2 = 16 remainder 0

16 ÷ 2 = 8 remainder 0

8 ÷ 2 = 4 remainder 0

4 ÷ 2 = 2 remainder 0

2 ÷ 2 = 1 remainder 0

1 ÷ 2 = 0 remainder 1

8 bit representation of 'A' – 01000001 –

add ‘0’on left side of binary code to make all binary values the same bit-length. Computers store data in **fixed sizes**: 8 bits, 16 bits, 32 bits, etc.  
So if a binary number is shorter, we add **leading zeros** to fill up the size.

16 bit representation of 'A' - 00000000 01000001 – add 8 ‘0’ ‘s.

Task 4:

What are static and dynamic arrays? Explain or summarize key points in a table like

Size, performance, memory, flexibility, limitations

| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | Fixed when created | Grows/shrinks as needed |
| **Performance** | Faster (less overhead) | Slightly slower (needs resizing/copying) |
| **Memory** | Continuous & fixed memory block | Allocates more memory when full |
| **Flexibility** | Not flexible — can’t change size | Very flexible — can grow or shrink |
| **Limitations** | Wastes memory if too big or overflows | May need copying, resizing (performance hit) |

### Static Array:

* Allocated a **fixed size** in memory at compile time.
* Cannot grow once created.
* Example in Java: int[] arr = new int[5];

Eg: int[] nums = new int[3];

nums[0] = 10;

nums[1] = 20;

### Dynamic Array:

* Grows or shrinks at **runtime**.
* Allocates **extra space** to reduce frequent resizing.
* Java uses ArrayList, Python uses list.
* Example in Java: ArrayList<Integer> list = new ArrayList<>();

Eg: import java.util.ArrayList

ArrayList<Integer> nums = new ArrayList<>();

nums.add(10);

nums.add(20);

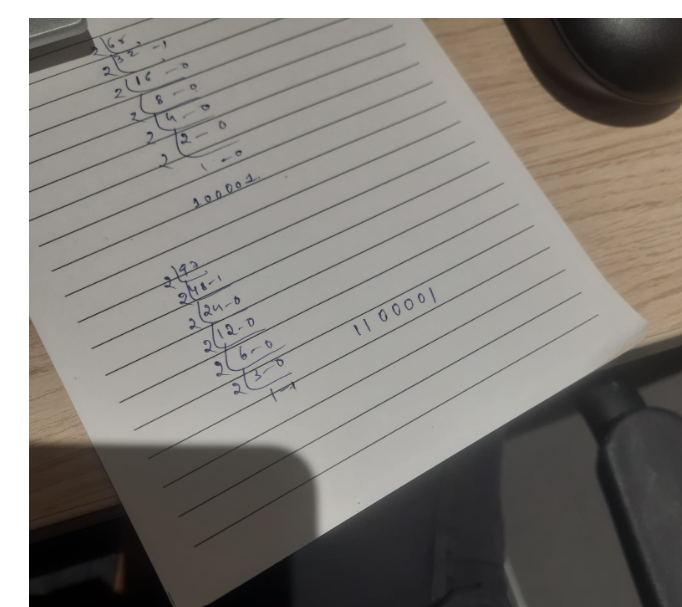
nums.add(30);

Task 5:

What is the binary value of a?

Hint ascii value is 97..

1100001



Task 6:

Types of Computer memory with examples.. Explain ..

### 1. ****Primary Memory (Main Memory)****

* Directly accessed by the **CPU**
* Faster but **limited in size**
* Temporarily stores data while the computer is running

| **Type** |  | **Full Form** | **Use** | **Volatile?** | **Example** |
| --- | --- | --- | --- | --- | --- |
| **RAM** |  | Random Access Memory | Stores running programs and data | Yes | 8GB DDR4 RAM |
| **ROM** |  | Read Only Memory | Stores firmware (like BIOS) | No | Boot instructions |
| **Cache** |  | – | Speeds up CPU by storing frequent data | Yes | L1, L2, L3 cache |
| **Registers** |  | – | Tiny memory inside CPU | Yes | Used in calculations |
| 2. ****Secondary Memory (Storage)****  * Stores data **permanently** * Slower than primary memory * Not directly accessed by CPU (needs I/O)  | **Type** | **Description** | **Example** | | --- | --- | --- | | **Hard Disk (HDD)** | Magnetic storage | 1TB Seagate HDD | | **Solid State Drive (SSD)** | Fast flash storage | 512GB NVMe SSD | | **CD/DVD** | Optical storage | CD with songs | | **Pen Drive** | Portable flash storage | 16GB USB drive | | **Memory Card** | Small flash storage | 32GB SD card | |  |  |  |  |  |

**Eg:**

* **RAM**: Like your notebook — you write temporarily while working.
* **ROM**: Like printed rules in a manual — can't easily change.
* **Cache**: Sticky notes for quick access.
* **Hard Disk**: A big locker where you store your stuff.
* **Pen Drive**: A portable suitcase you carry between places.

**Linked list**

**data**

**Address of previous node Address of next node**

**Whenever we insert, positions change so, index numbers changes.**

**Whenever we delete, position shifts.**

A **Linked List** is a way to **store a list of items** where each item **points to the next one**.

Eg: Imagine a **chain of boxes**, where:

* Each box = a **node**
* Each node has: **Data** (the value), A **link** (pointer) to the next box

Unlike arrays, **linked lists don’t need to be next to each other in memory** — they just point to the next item.

## Types of Linked List

| **Type** | **Simple Meaning** | **Diagram Style** |
| --- | --- | --- |
| **Singly Linked List** | Each node points to **next** only | A → B → C → null |
| **Doubly Linked List** | Each node points to **next and previous** | null ← A ⇄ B ⇄ C → null |
| **Circular Linked List** | Last node connects to **first node** | A → B → C → A |
| **Doubly Circular List** | Nodes connected in both directions and **loops back** | A ⇄ B ⇄ C ⇄ A |

**Masked memory**

**Masked memory** is mostly used in **ROM (Read-Only Memory)**, specifically **masked ROM**, where the data is permanently written **at the time of manufacturing**.

**Cannot be changed or erased** after being made.

**Eg: Printers,** Microcontrollers

| **Feature** | **Masked Memory (Masked ROM)** |
| --- | --- |
| Can be changed? | No |
| When is data written? | manufacturing |
| Used in? | Embedded devices, BIOS, hardware firmware |
| Cost-effective? | Yes, but only in bulk |

**PROM**

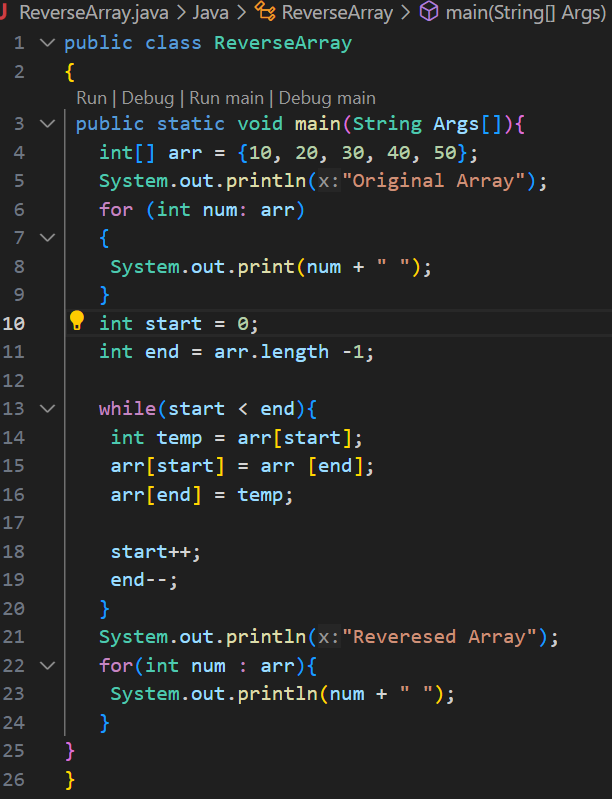
**PROM (Programmable Read-Only Memory)** is a type of memory chip that can be **programmed only once** by the user **after manufacturing**.

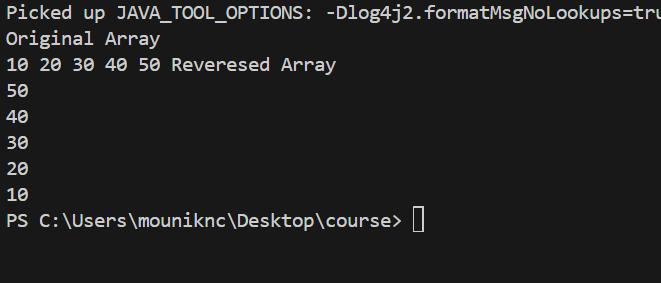
When PROM is made, it’s **empty** (unprogrammed). You can program it **once** using a special device called a **PROM programmer**. After that, the data is **permanently stored** and **cannot be changed or erased**. Each bit is programmed by **burning fuses** inside the chip.

Eg: you get a **blank scratch card** 🃏: You can scratch and reveal numbers **only once**. After that, it’s **locked forever** — can’t be reused.

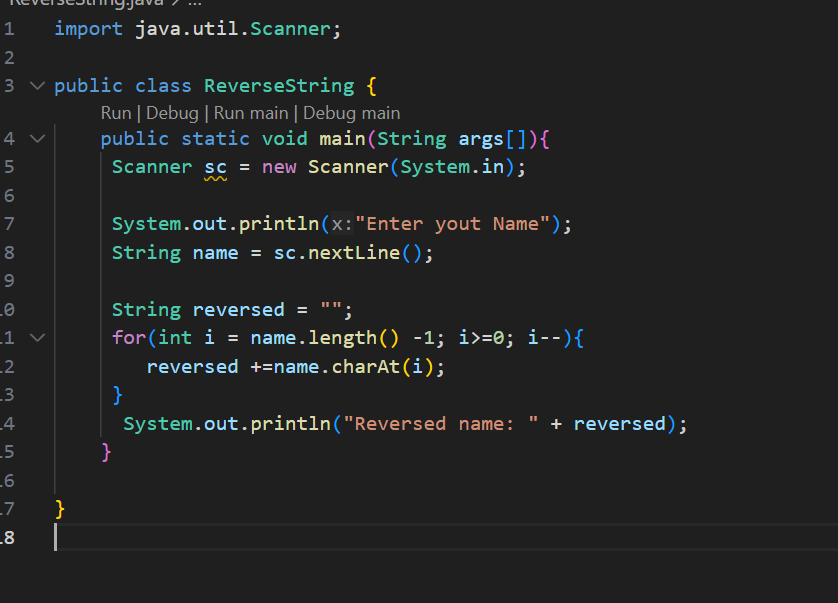
| **Feature** | **PROM** |
| --- | --- |
| Full form | Programmable Read-Only Memory |
| Writable? | Only once |
| Erasable? | No |
| Used for? | Permanent firmware, chips |
| Programmed by? | PROM burner / programmer |

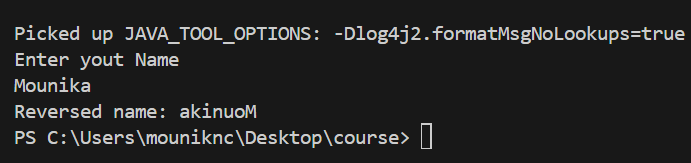
**Task 7: simple code to reverse an array**

****

****

**Task 8: Reverse a string .. write a code. Hint: take a name from the user and display the name in reverse order..**

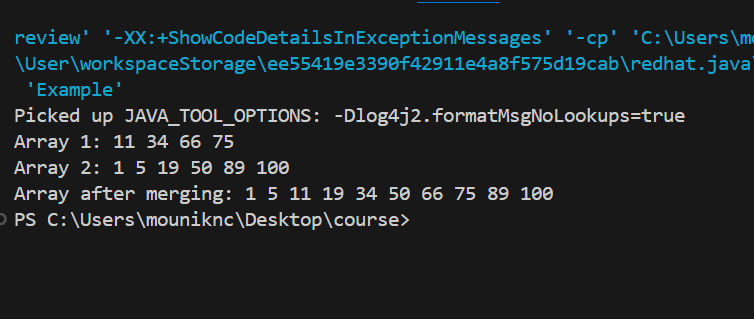
****

****

**Task 9: Website registration: No**

**Task 10:**

public class Example {  
   public static void main (String[] args) {  
      int[] arr1 = {11, 34, 66, 75}; // this is the first sorted array  
      int n1 = arr1.length;  
      int[] arr2 = {1, 5, 19, 50, 89, 100}; // this is the second sorted array  
      int n2 = arr2.length;  
      int[] merge = new int[n1 + n2]; // merge[10] -> stores final sorted result  
      int i = 0, j = 0, k = 0, x;  
      System.out.print("Array 1: ");  
      for (x = 0; x < n1; x++) // print array  
      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) { // comparing elements from both arrays, put the smaller one into merge[]   
         if (arr1[i] < arr2[j]) // keep doing until 1 array is finished  
            merge[k++] = arr1[i++];  
         else  
            merge[k++] = arr2[j++];  
      }  
      while (i < n1) // If one array finishes first, the rest of the other array is added directly.  
      merge[k++] = arr1[i++];  
      while (j < n2)  
      merge[k++] = arr2[j++];  
      System.out.print("\nArray after merging: ");  
      for (x = 0; x < n1 + n2; x++) // printing final merged array  
      System.out.print(merge[x] + " ");  
   }  
}

****

**Home task:** Task 10  rewrite the code in such a way that it has to take unsorted list and then ,merge in an array the sorted list.

Hash Tables stores key-value data and is used to search data quickly using key.

Task 11:

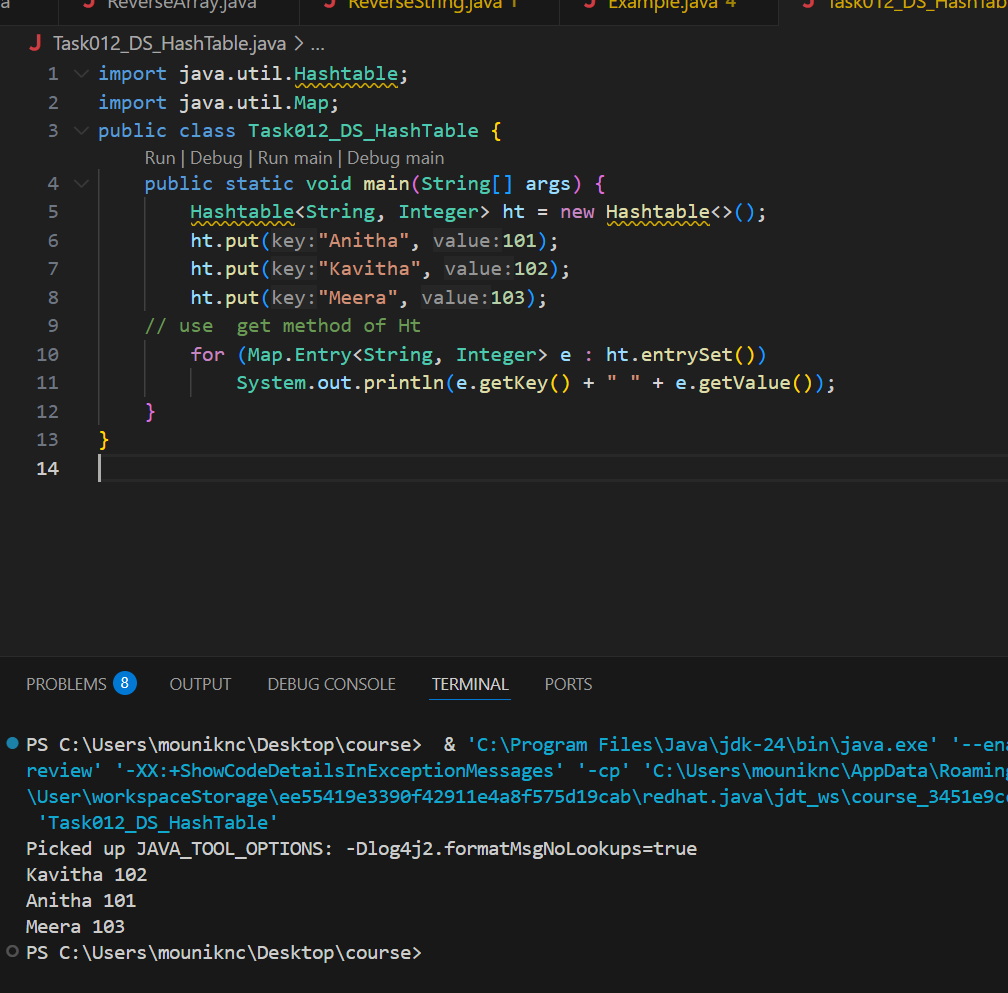
What do you understand by Hash table?

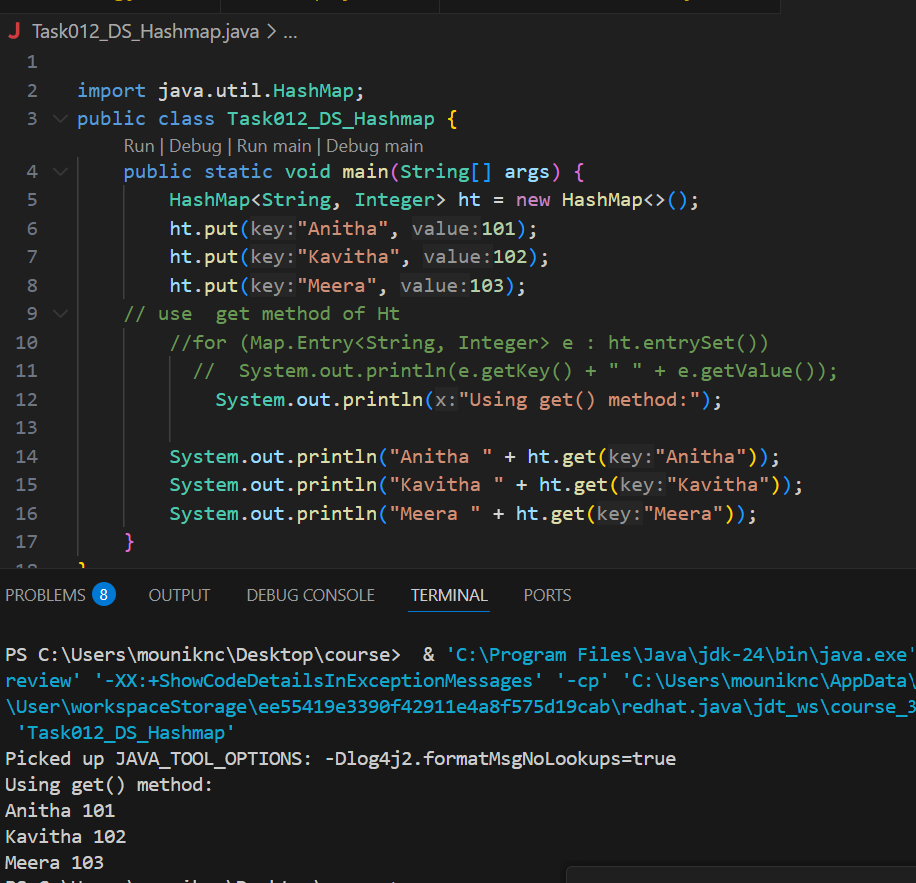
**A Hash Table** **is a** **data structure** **that stores data in** **key-value pairs**, **and it uses a** **hash function** **to find things** **very fast**.

**Eg: you have a locker with 100 boxes**, **and each box has a number.  
Now, let’s say you want to store your PAN card.  
Instead of randomly opening lockers, you use a function (like your name or ID) to find the exact locker**.

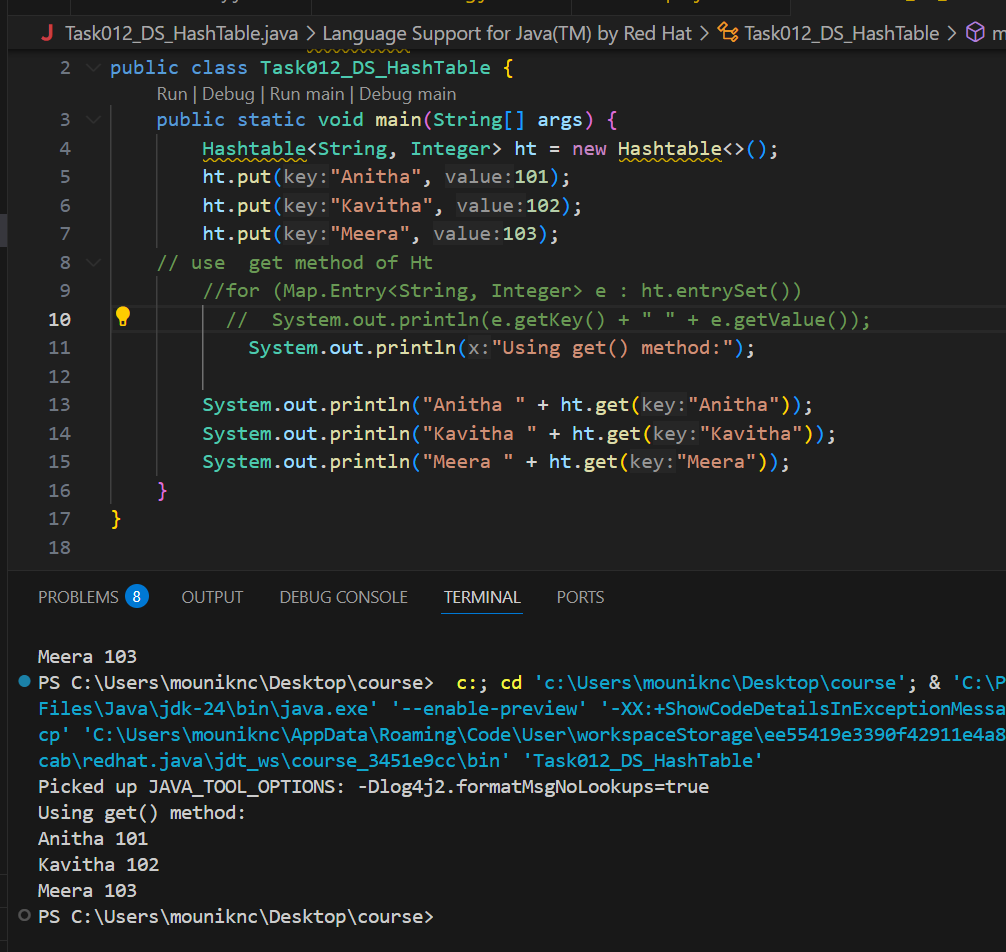
**That’s what a Hash Table does.**

**Task 12:**

****



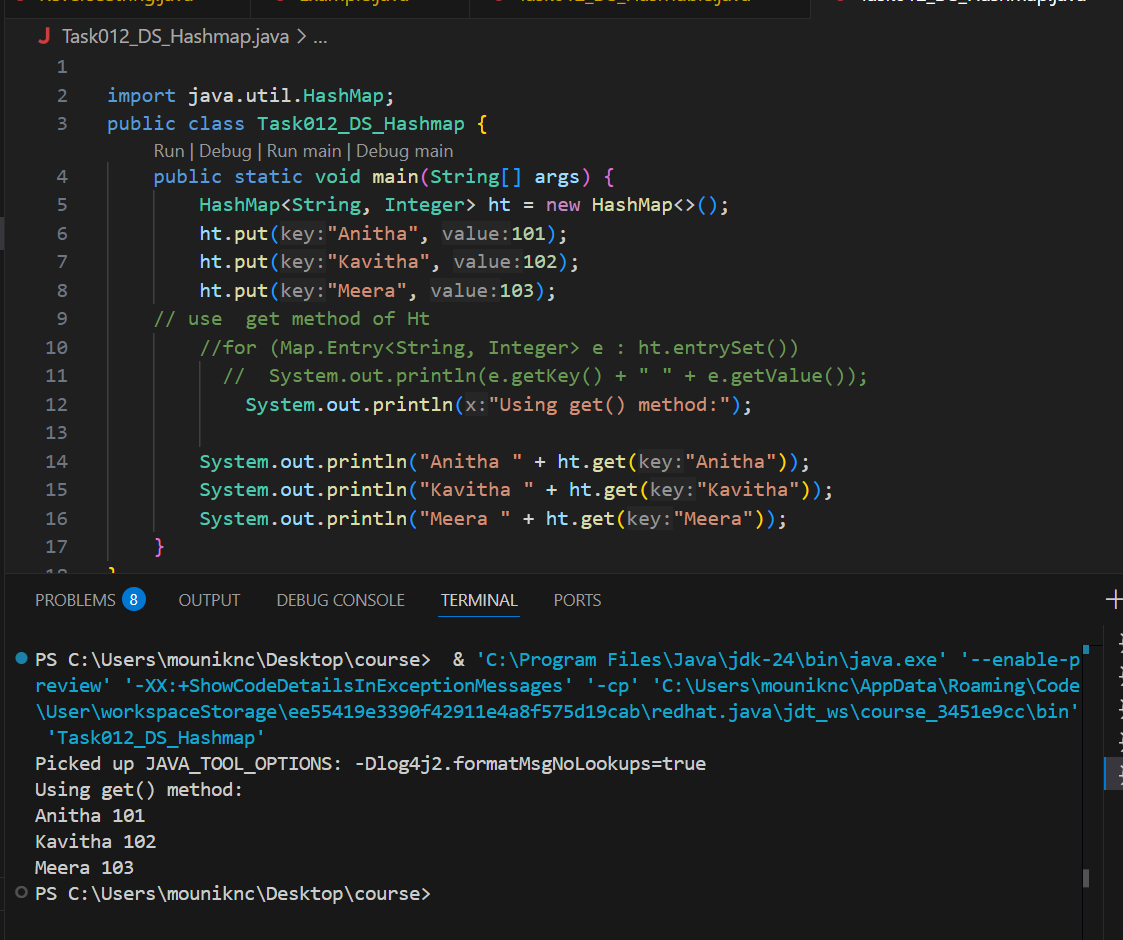
Understand the below Hash table code and try to print values using get method of Hash table



HashTable – Synchronized. – Thread safe – Can access only 1 thread at a time. – It is slower – It won’t take Null key & value

HashMap – Asynchronized. – Not Thread safe – Can access multiple threads at a time. N- It is faster – It will take one null key and multiple null values.

Task 13: Wap to create  a hash map and display them..



Task 14: Difference between Hash Table and Hash Map

|  |  |
| --- | --- |
| **HashTable** | **HashMap** |
| It is Synchronized | It is Asynchronized |
| Internally linked list is present | Internally linked list is present |
| Thread Safe | Not Thread safe |
| Can access only 1 thread at a time | Can access multiple threads at a time |
| It won’t take Null key & value | It will take one null key and multiple null values. |
| It is slower | It is faster |

Similarities

hash table and Hash Map have linked list internally.

Collisions occur in Hash Table and hash Maps.

Collision in Hash map can handle separate chaining, Open addressing etc..

Where do we use set, put  
//set -- arrayList , replace the values , updates the previous value

//put -- hash table , insert the value, puts a new value

Task 15: Linear probing in Hash table

**Linear Probing** is a **collision resolution** technique used in hash tables.  
If the spot at the hash index is taken, it checks the **next index (index + 1)**, and so on — **linearly** — until an empty spot is found.

import java.util.Scanner;

class LinearProbingHashTableCode {

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 temp = hash(key);

int i = temp;

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 != temp);

}

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 temp1 = keys[i], temp2 = vals[i];

keys[i] = vals[i] = null;

currentSize--;

insert(temp1, temp2);

}

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 HashTableLinearProbing

{

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");

LinearProbingHashTableCode lpht = new LinearProbingHashTableCode(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;

}

// Display hash table

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:

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

