Day 12

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

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?

2 min

01000001

**Task 2:**

What is the binary value of a?

2min

a = 97

97/2 = 1

48/2 = 0

24/2 = 0

12/2 = 0

6/2 =0

3/2 = 1

1

binary value of a = 01100001

Hint ascii value is 97..

10.46 to 10.48

**Task 3:**

Types of Computer memory with examples.. Explain ..

11.04 to 11.14

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

### **a) RAM (Random Access Memory)**

* **Temporary** memory used while the computer is ON.
* Stores data and programs **currently in use**.
* Data is lost when power is off.

**Example**:

* Running apps like MS Word, a game, or a web browser.

Types of RAM:

* **DRAM** – Dynamic RAM (needs frequent refreshing)
* **SRAM** – Static RAM (faster, used in cache)

### **b) ROM (Read-Only Memory)**

* **Permanent** memory; cannot be modified easily.
* Contains **booting instructions** (e.g., BIOS).
* Data is **not lost** when power is off.

**Example**:

* ROM in a washing machine’s control system
* BIOS in your computer (Basic Input/Output System)

Types of ROM:

* **PROM** – Programmable once
* **EPROM** – Erasable using UV light
* **EEPROM** – Electrically erasable (used in BIOS chips)

## **2. Secondary Memory (Storage memory)**

* Used for **long-term storage** of files, software, OS, etc.
* **Slower** than primary memory but **permanent**.
* Data stays even when the power is off.

### **Examples:**

| **Storage Type** | **Real-World Example** |
| --- | --- |
| **HDD** (Hard Disk) | Laptop storage (C: drive) |
| **SSD** | Faster storage in modern computers |
| **CD/DVD** | Media storage (music, movies) |
| **USB Drive** | Portable file storage |
| **Memory Cards** | Used in cameras, phones |

## **3. Cache Memory**

* Very **fast memory** inside the CPU.
* Stores **frequently used data** to speed up processing.
* Faster than RAM, but smaller in size.

**Example**:

* A CPU quickly accessing recently opened files or browser tabs.

## **4. Virtual Memory**

* A **portion of hard drive** used as extra RAM when real RAM is full.
* Slower than actual RAM but helpful for multitasking.

**Example**:

* Your system doesn’t crash even when you open many tabs — thanks to virtual memory.

## 

| **Memory Type** | **Speed** | **Volatile?** | **Used For** | **Example** |
| --- | --- | --- | --- | --- |
| RAM | Fast | Yes | Running programs | MS Word, Browsers |
| ROM | Slow | No | Boot process, firmware | BIOS |
| Cache | Very Fast | Yes | Temporary CPU access | CPU Instructions |
| Virtual Memory | Slow | Yes | Extra RAM from disk | Paging files |
| Secondary Storage | Slower | No | Saving files and data | SSD, HDD, USB |

**Task 4:**

What do you understand by data structures..?

Data structures are strategies for handling data efficiently like choosing the right storage or path based on what you need speed, flexibility, size, or rules.

You wouldn’t keep a contact list in a photo album, right?

Similarly, in programming, choosing the right data structure makes your code faster, cleaner, and smarter.

11.46 to 11.50

Task 5:

What are the operations on data structures ?

### **1. Insertion**

**Adding** a new element

* Add a student to a class list
* Add a message to your inbox

Example: arr[3] = 10;

### **2. Deletion**

**Removing** an element

* Delete a contact from your phone
* Remove the last task from a to-do list

Example: arr[3] = NULL;

### **3. Traversal**

**Visiting all elements** one by one

* Reading each file in a folder
* Checking each number in a list

Example (pseudocode):

for(i = 0; i < n; i++) {

print(arr[i]);

}

### **4. Searching**

**Finding** an element in the structure

* Find a name in your contact list
* Search a file on your computer

Example: if(arr[i] == 20) {

### 

### 

### 

### **5. Sorting**

**Arranging** elements (ascending or descending)

* Sort students by marks
* Sort files by date or name

Example: sort(arr);

### **6. Updating**

**Changing** an element’s value

* Update a contact number
* Change a task’s priority

Example: arr[2] = 45;

| **Operation** | **Purpose** | **Real Example** |
| --- | --- | --- |
| Insertion | Add element | Add student to a list |
| Deletion | Remove element | Remove item from cart |
| Traversal | Visit all elements | Go through all emails |
| Searching | Find element | Search contact by name |
| Sorting | Arrange elements | Sort marks high to low |
| Updating | Change existing data | Edit profile info |

11.51 to 11.53

Task 6:

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

Size, performance, memory, flexibility

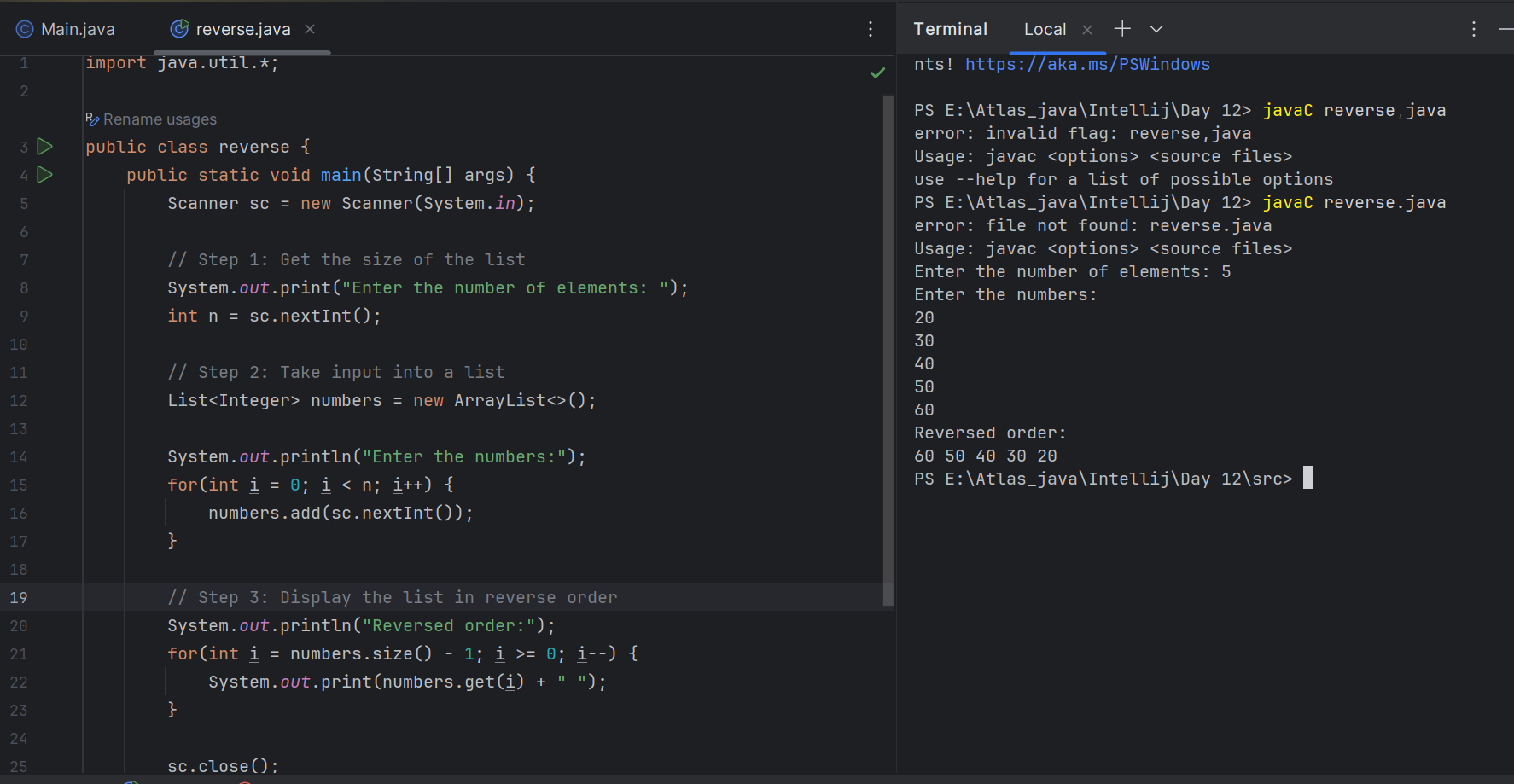
| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | Fixed at compile time | Flexible, can be changed at runtime |
| **Performance** | Faster (no allocation overhead) | Slower (due to resizing/reallocation) |
| **Memory** | Allocated on **stack** (limited) | Allocated on **heap** (more available) |
| **Flexibility** | Low – can’t resize or shrink | High – can grow or shrink as needed |

**Task 7:**

Reverse an array. write a code.

11.59 to 12.02

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

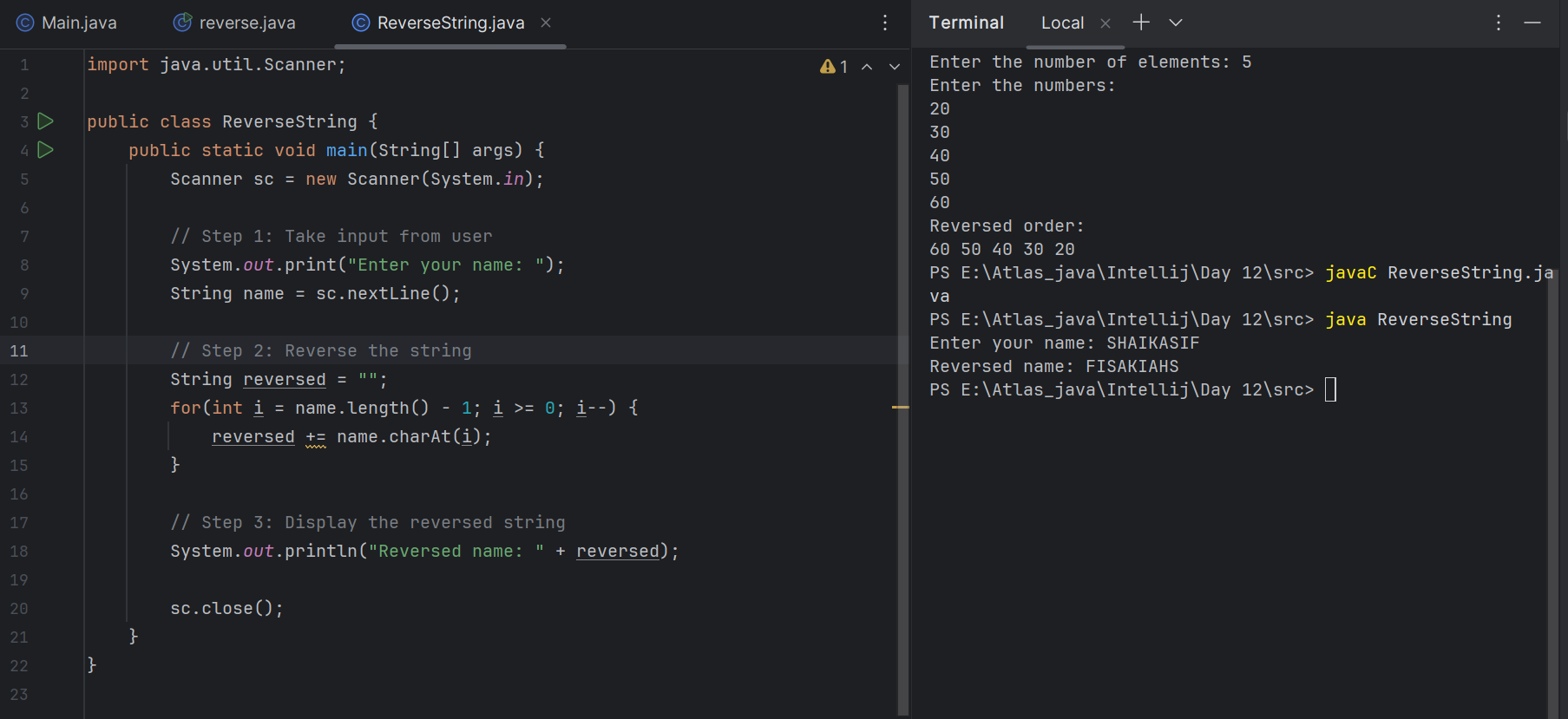


**Task 8:**

Reverse a string .. write a code.

12.03 to 12.06

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



**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] + " ");

}

}

12.11 to 12.15 to 12.17

What is the above code snippet doing..?

* arr1 = {11, 34, 66, 75} (Sorted)
* arr2 = {1, 5, 19, 50, 89, 100} (Sorted)

### Merging Logic:

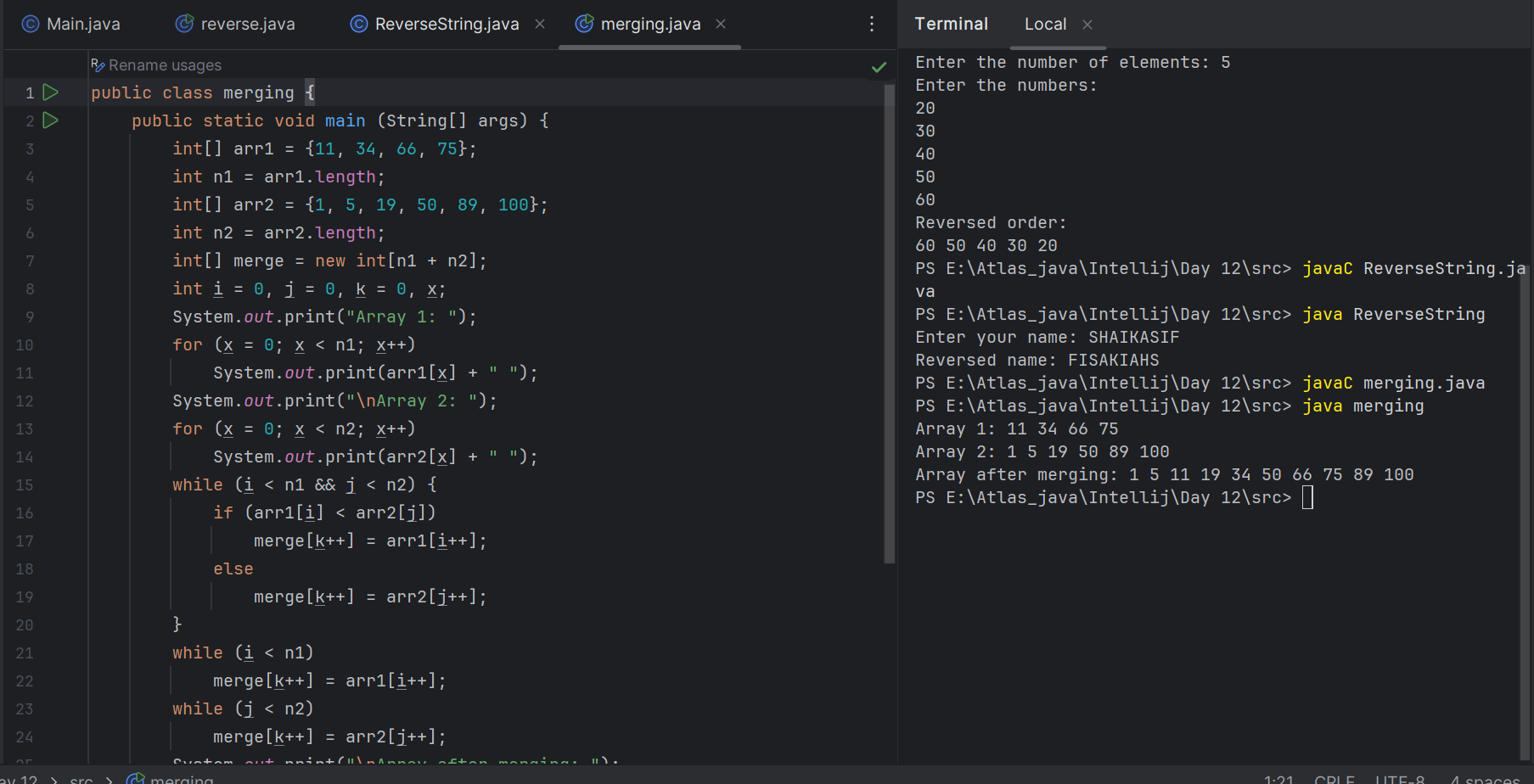
* Uses two pointers i and j to traverse both arrays.
* Compares elements from both arrays:  
  + If arr1[i] < arr2[j], it adds arr1[i] to merge[].
  + Otherwise, adds arr2[j] to merge[].
* Continues until all elements are merged into the new array merge[].

### Output:

### Array 1: 11 34 66 75

### Array 2: 1 5 19 50 89 100

### Array after merging: 1 5 11 19 34 50 66 75 89 100



### 

### 

**Task 11:**

What do you know about hash table?

A hash table is a data structure that stores key-value pairs. It’s designed to quickly find, insert, or delete data.

Think of it like a dictionary or contact list:

* You look up a name (the key),
* And you get the phone number (the value).

**Key Concepts**

| **Concept** | **Explanation** |
| --- | --- |
| **Key** | What you search with (like a name) |
| **Value** | The data stored (like a phone number) |
| **Hash Function** | A formula that converts the key into an **index** (location) in an array |
| **Bucket** | The place where the value is stored (could hold multiple if there's a collision) |
| **Collision** | When two keys hash to the same index (handled using chaining or probing) |

**Basic Operations**

| **Operation** | **What it does** | **Time Complexity (Avg)** |
| --- | --- | --- |
| insert(k, v) | Add key k with value v | O(1) |
| get(k) | Find value by key | O(1) |
| delete(k) | Remove key-value pair | O(1) |

## **Where Hash Tables Are Used:**

* Databases – for indexing
* Compilers – for symbol tables
* Caching systems
* Sets and Maps in most programming languages
* Password storage (with secure hash functions)

12. 18 to 12. 22

**Task 12:**

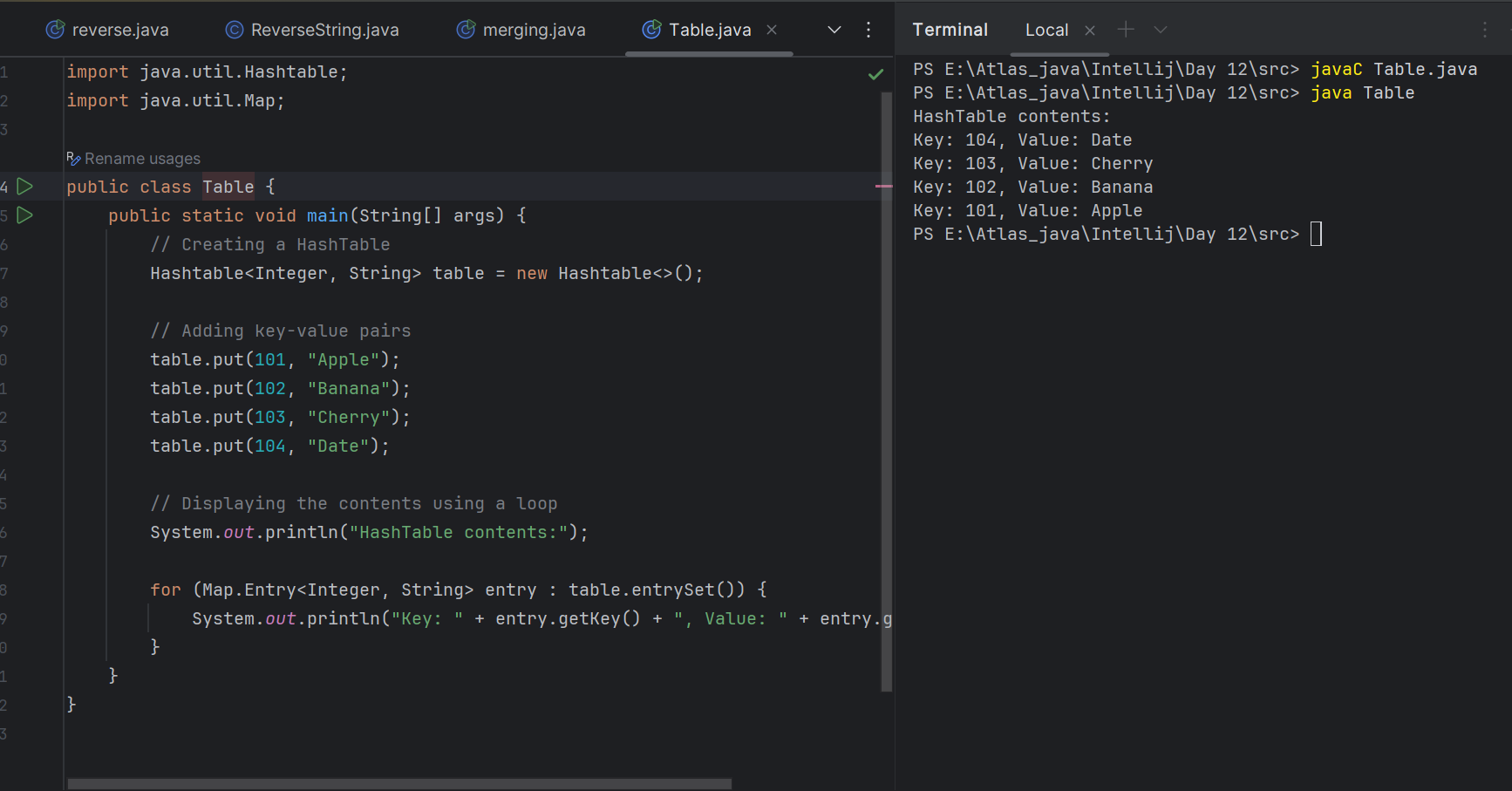
Wap to create a hash table and display them..

Hint 👍

Import java.util.Hashtable;

Import java.util.Map;

12.39 to 12.44



// Map is an interface

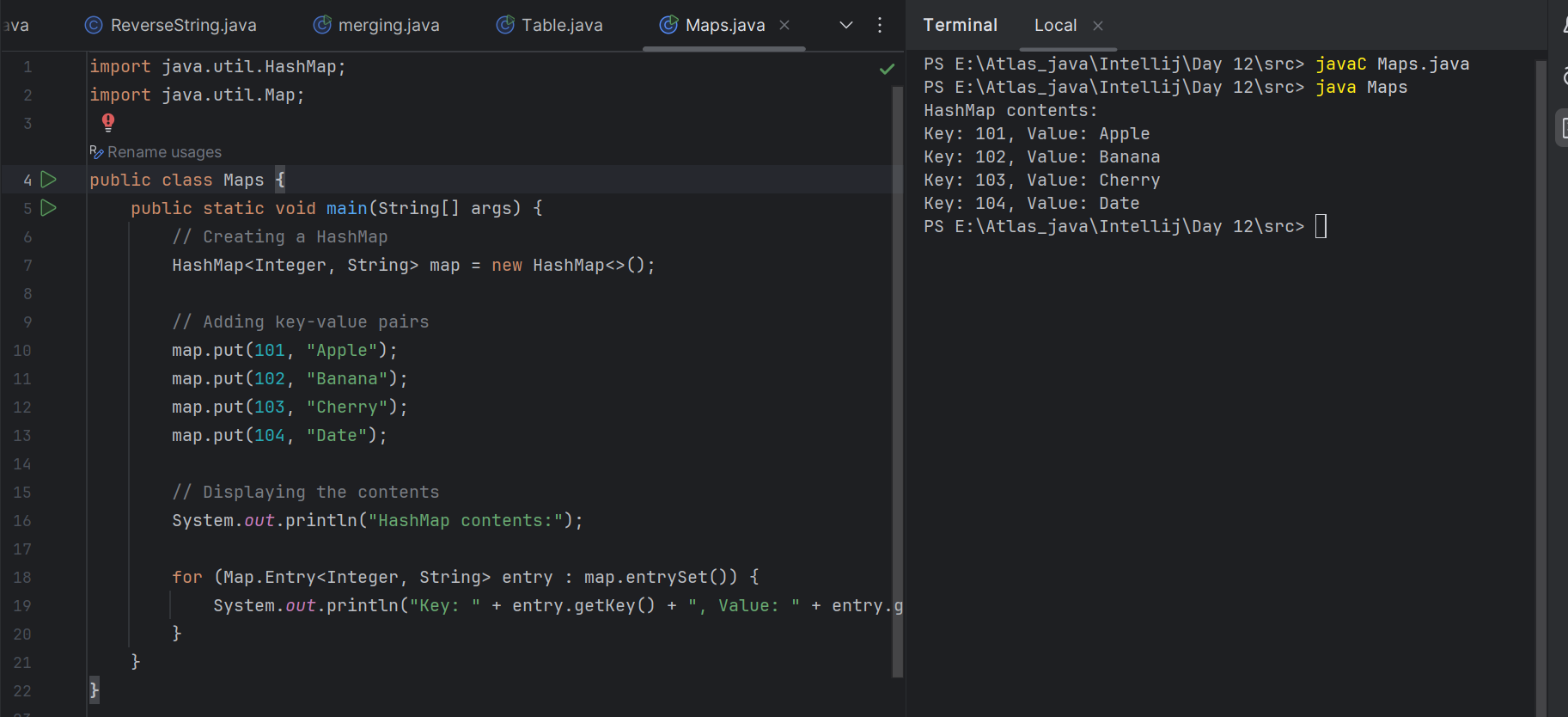
// hash table --> slower , sync , thread safe, no null value accepted

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

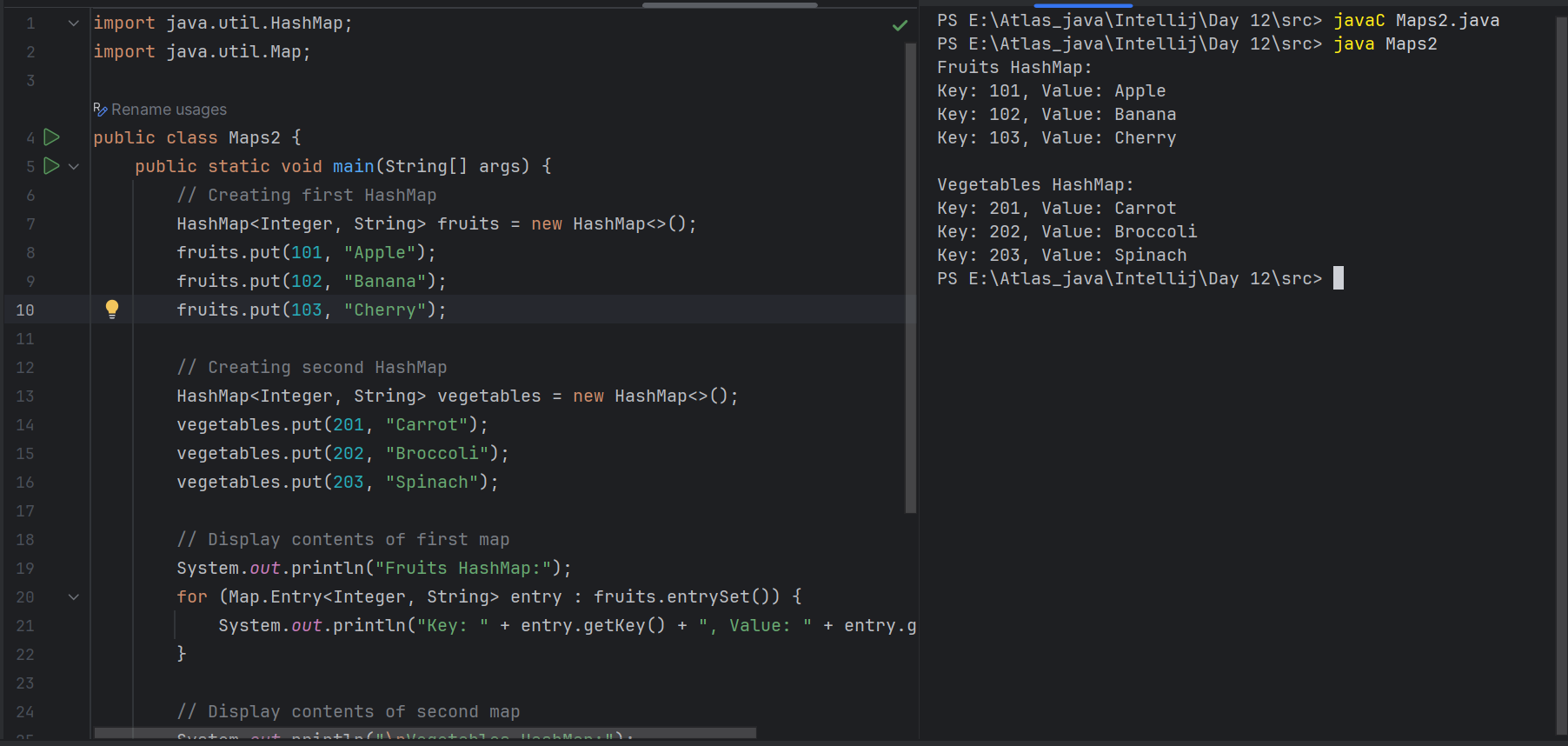
**Task 13:**

Wap to create a hash map and display them..

By using HashMap:



Creating with 2 objects



//set -- arrayList , replace the values , updates the previous value

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

**Task 14:**

Hash table advantages and disadvantages

4 min 13.22 to 13.26

## **Advantages of Hash Tables:**

| **Advantage** | **Explanation** |
| --- | --- |
| **Fast Access (O(1))** | Average time to insert, delete, or search is constant, regardless of dataset size. |
| **Efficient Key-Value Storage** | Ideal for pairing data, e.g., usernames with passwords, employee IDs with names. |
| **Flexible Keys** | Can use integers, strings, or even custom objects (with correct hashing). |
| **Scalable** | Handles large datasets efficiently. |
| **Built-in in Most Languages** | Easy to use: HashMap (Java), dict (Python), unordered\_map (C++). |

## **Disadvantages of Hash Tables:**

| **Disadvantage** | **Explanation** |
| --- | --- |
| **No Order Guarantee** | Elements are not stored in a predictable order. |
| **Collisions** | Two keys might hash to the same index, which requires special handling (e.g., chaining or open addressing). |
| **Inefficient Memory Use** | Can use more memory than necessary due to internal resizing or gaps in storage. |
| **Hash Function Dependency** | Bad or poorly designed hash functions can slow down performance drastically. |
| **Not Suitable for Range Queries** | You can’t easily find all keys between two values like in a tree structure. |

Lunch 1.30 to 2.30

Linear probing in Hash table

**Task15:**

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

public class Main {

public static void main(String[] args) {

HashTable<String, String> ht = new HashTable<>();

ht.put("101", "Alice");

ht.put("102", "Bob");

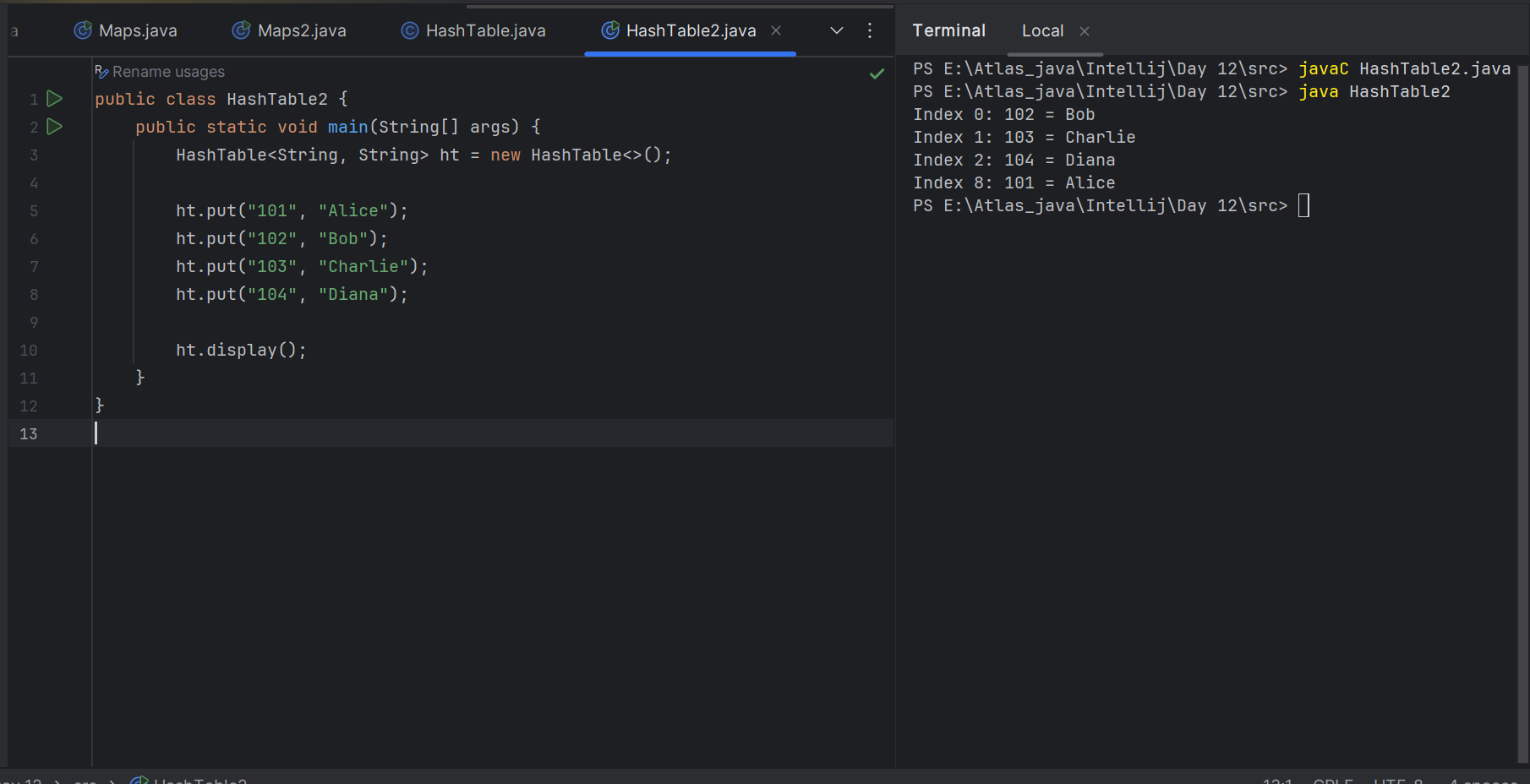
ht.put("103", "Charlie");

ht.put("104", "Diana");

ht.display();

}

}



8 min 15.08 to 15.16

**Task 16:**

Methods of Hash table plz list them..

| **Method** | **Description** |
| --- | --- |
| put(K key, V value) | Inserts or updates a key-value pair. |
| get(Object key) | Returns the value for the specified key, or null if not found. |
| remove(Object key) | Removes the entry for the specified key. |
| containsKey(Object key) | Checks if the table contains the specified key. |
| containsValue(Object value) | Checks if the table contains the specified value. |
| isEmpty() | Returns true if the table contains no key-value mappings. |
| size() | Returns the number of key-value mappings. |
| clear() | Removes all mappings (empties the hashtable). |
| keySet() | Returns a Set view of the keys. |
| values() | Returns a Collection view of the values. |
| entrySet() | Returns a Set view of the key-value mappings. |
| clone() | Creates a shallow copy of the hashtable. |
| equals(Object o) | Compares two hashtables for equality. |
| hashCode() | Returns the hash code value for this hashtable. |

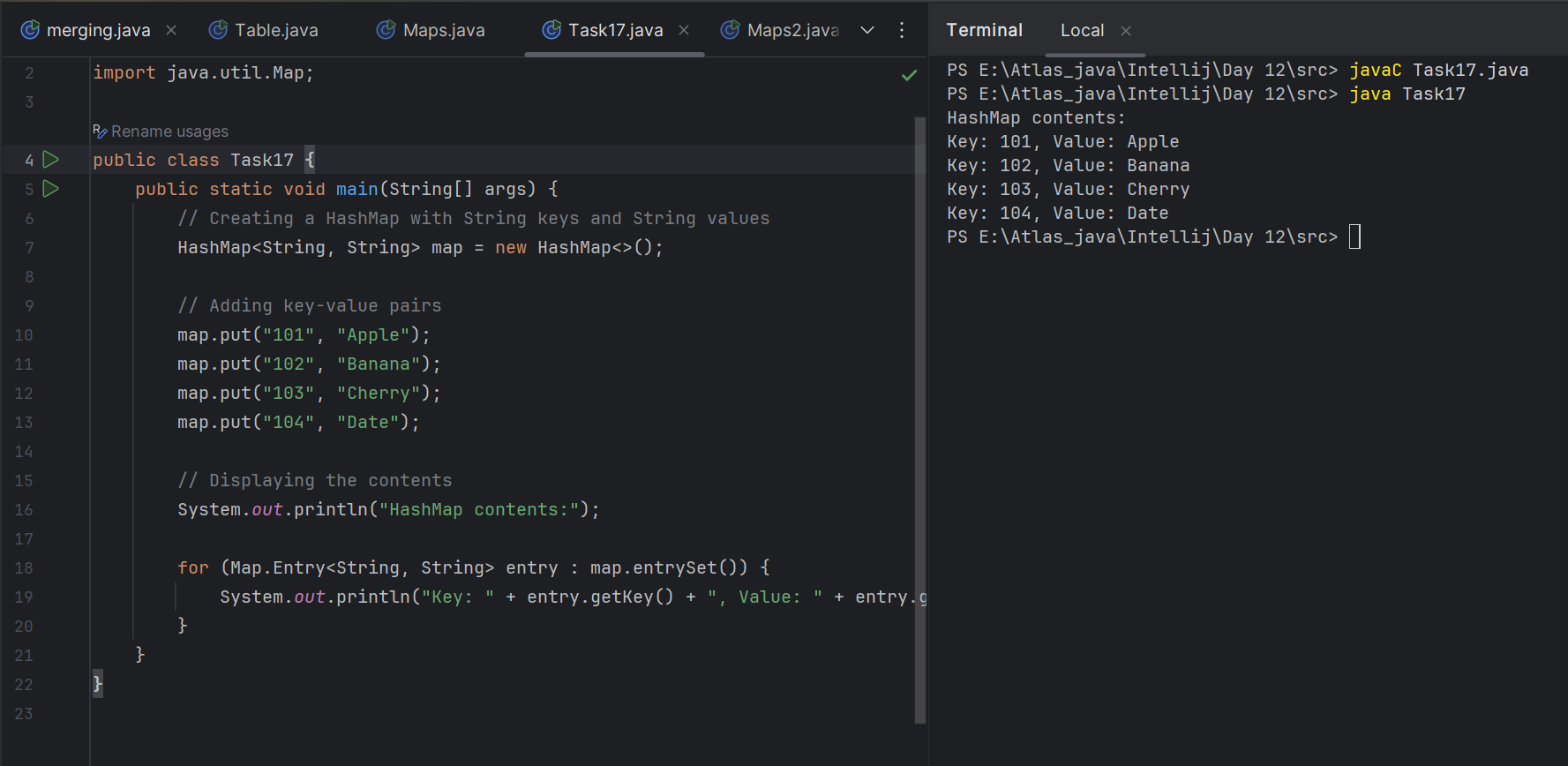
**Task 17:**

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

Like HashMap<Integer, String>

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

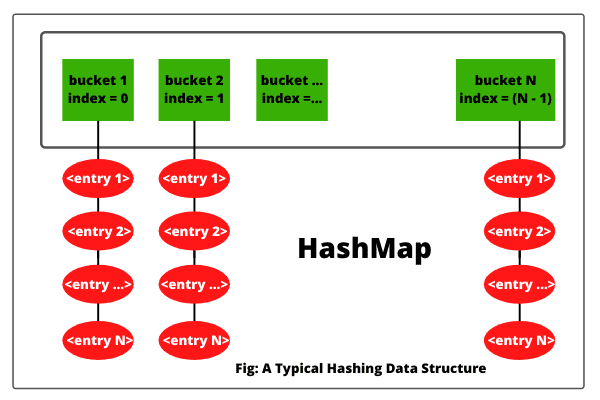
Like this HashMap<String, String> ? will this work?



**It is working.**

Task 18:

### Explain the internal working of a HashMap. With diagram..

****

**Buckets (Green boxes):**  
 Represent the underlying array of the HashMap. Each bucket is a slot/index in the array.  
 Example: table[0], table[1], ..., table[n-1].

**Entries (Red ovals):** These are the key-value pairs stored in each bucket.

If multiple keys hash to the same index (bucket), they are stored in a LinkedList or Tree structure this is called collision handling.

(Simply we say two keys may hash to the same bucket. This is called a collision.)

**Home task:**

**Do hash table have linked list internally?**

Yes Hash tables *can* use Linked Lists internally, depending on the implementation.

**Do collisions occur in hash Maps?**

Yes, collisions do occur in HashMaps. Collisions in HashMap occur when two or more different keys are hashed to the same index (bucket) in the internal array..

They are :

Hash collision: When hashCode() of two different keys are the same,

Bucket collision: When two keys have different hashCodes but land in the same bucket due to (hash % array.length)

| **Key** | **Hash Code** | **Index (bucket)** |
| --- | --- | --- |
| "dog" | 98324 | 4 |
| "god" | 98324 | 4 |

**What is load factor and how the capacity increases?**

* Load Factor is a measure of how full the HashMap can get before it resizes (rehashes).
* It’s a threshold value between 0.0 and 1.0.

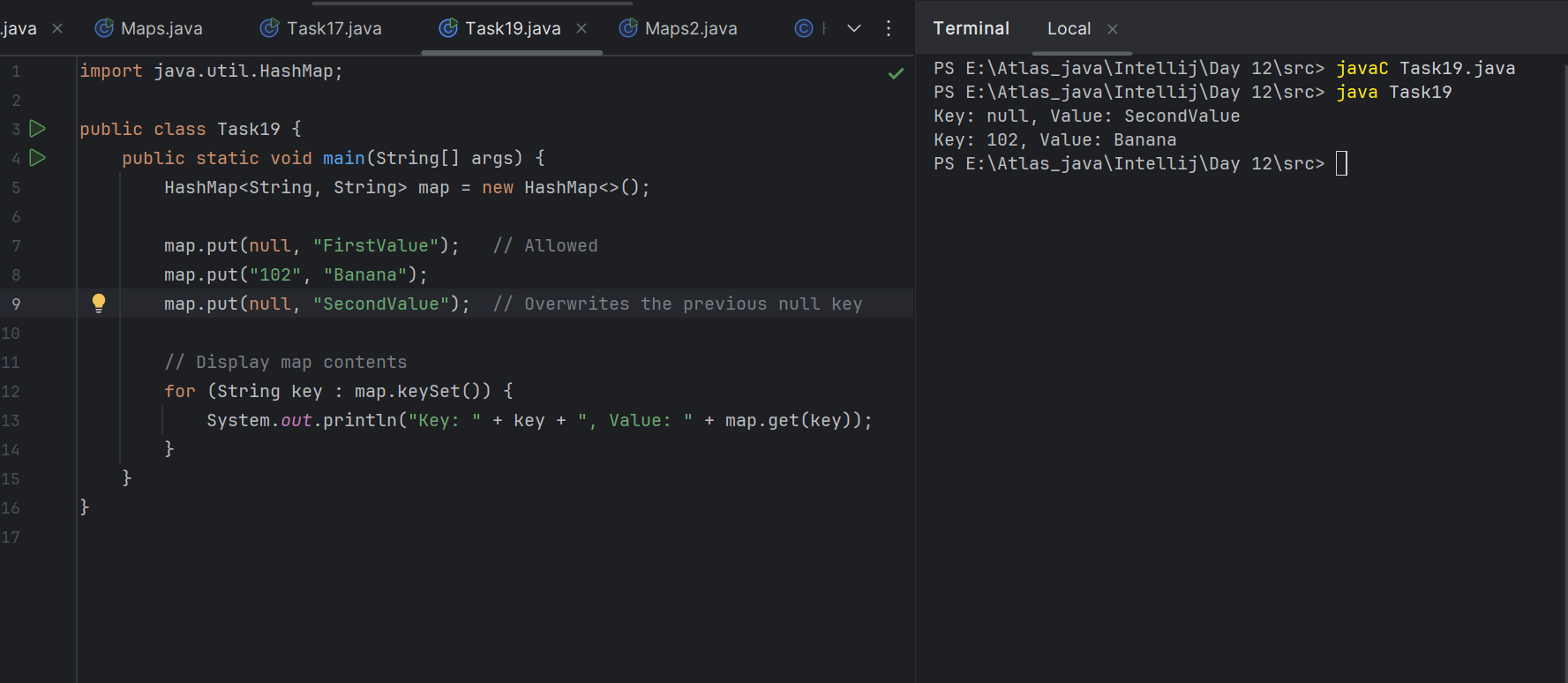
Threshold = Initial Capacity × Load Factor

**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..

17.19 to 17.21



**Task 20:**

How many methods are there to create a hash Map?

5 min -> 17.35 to 17.40

| **Method** | **Description** |
| --- | --- |
| **Default Constructor** | Creates an empty map with default capacity (16) and load factor (0.75). |
| **With Initial Capacity** | Creates a map with a specified initial capacity to reduce resizing overhead. |
| **With Capacity and Load Factor** | Allows setting both initial size and load factor to control resizing behavior. |
| **Copy from Another Map** | Copies all key-value pairs from an existing map into a new map. |
| **Anonymous Inner Class (Double Braces)** | Quick inline initialization using an instance initializer block. |
| **Immutable Map (Java 9+)** | Creates an unmodifiable map using Map.of() (up to 10 key-value pairs). |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

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, 0.5f);

Initial capacity ===10

Load factor === 0.75f

**TAsk21:**

**write a code using Collections.synchronizedMap()**

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

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class SynchronizedMapExample {

public static void main(String[] args) {

// Creating a regular HashMap

Map<String, String> regularMap = new HashMap<>();

// Wrapping the HashMap with synchronizedMap

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

// Adding key-value pairs

syncMap.put("Movie1", "Inception");

syncMap.put("Movie2", "Interstellar");

syncMap.put("Movie3", "The Dark Knight");

// Synchronized block required when iterating

synchronized (syncMap) {

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

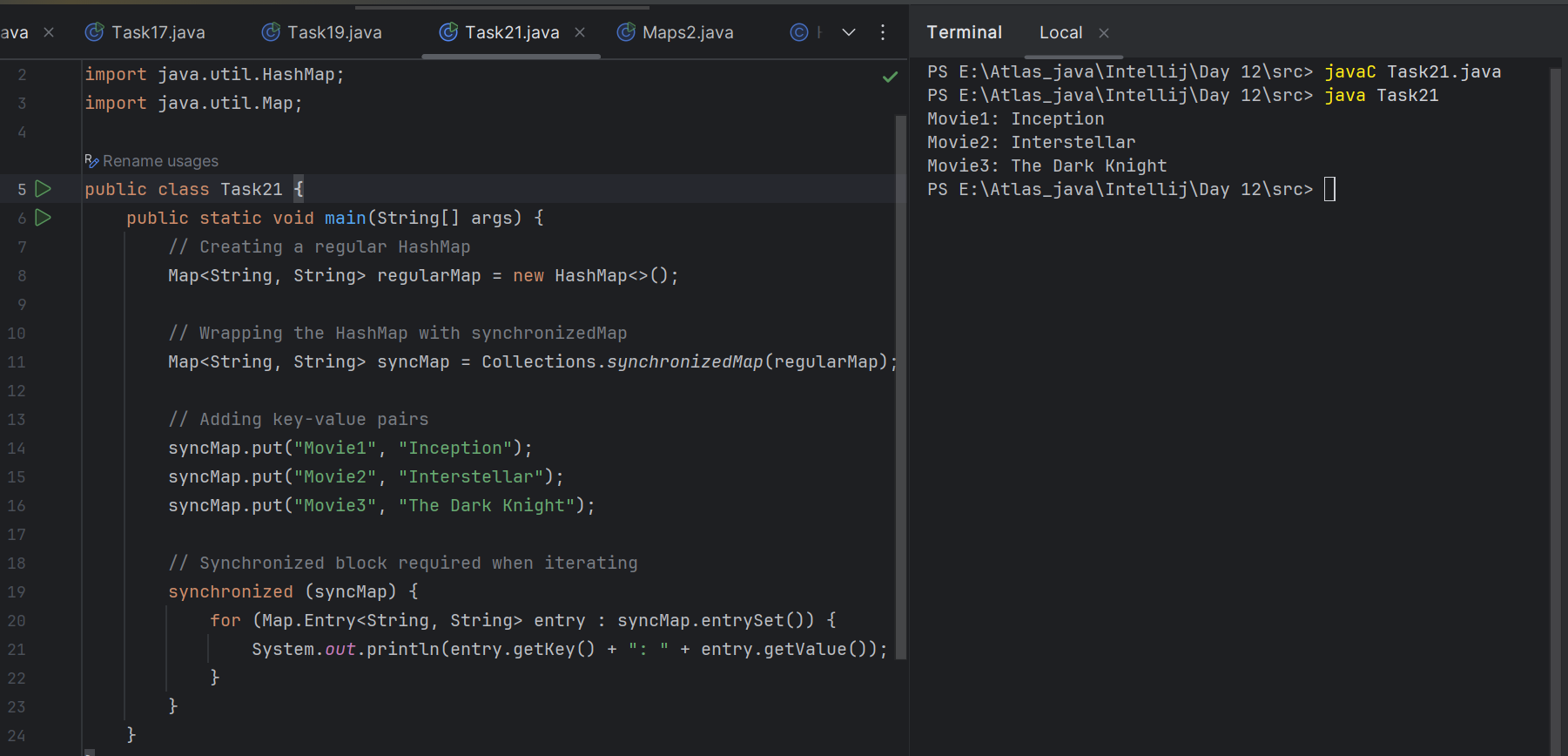
System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

}

****