**Array And Collection**

**1.Java Array In Depth**

**. Arrays in Java**

* An **array** is a collection of elements of the same data type, stored in contiguous memory locations.
* Arrays in Java are **objects**, meaning they are created dynamically at runtime.
* The size of an array is fixed once defined.
* Arrays are zero-indexed (the first element is at index 0).

**Corresponding Classes**

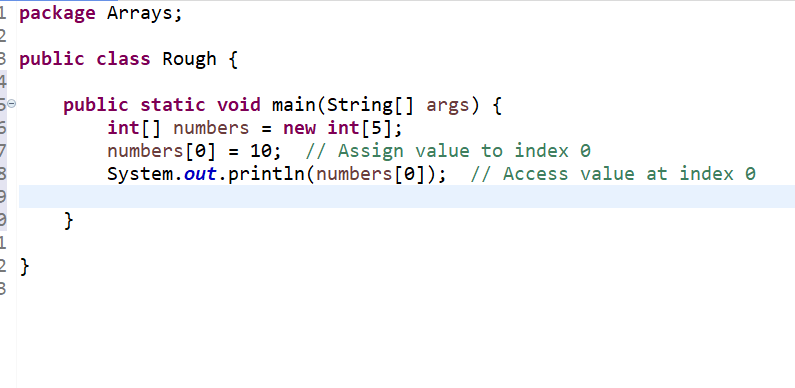
* Java arrays are part of the java.lang package and are internally represented as objects.
* They inherit directly from the Object class.

**2. One-Dimensional Array (1D Array)**

* A **1D array** is a linear collection of elements.

dataType[] arrayName = new dataType[size];

Example:



Initialization

int[] numbers = {1, 2, 3, 4, 5}; // Declare and initialize

Iteration:for (int i = 0; i < numbers.length; i++) {

System.out.println(numbers[i]);

}

**Key Points:**

* A 1D array is essentially a single row of elements.
* Can store data types like int, double, String, etc.

**Multidimensional Array**

A **multidimensional array** in Java is an array of arrays. Each array in the "parent" array can hold its own set of elements.

**Two-Dimensional Array (2D Array)**

A **2D array** is essentially a table (rows and columns).

**Example: int[][] matrix = new int[3][3];**

* The first 3 represents the **number of rows** in the array.
* The second 3 represents the **number of columns** in each row.
* Together, it creates a grid with 3 rows and 3 columns.
  + Total elements = 3 x 3 = 9

Here is how the structure looks:

Row 0: [0, 0, 0] // Default values

Row 1: [0, 0, 0]

Row 2: [0, 0, 0]

**Declaration and Initialization**

**1. Declaration and Initialization Using new:**

int[][] matrix = new int[3][3]; // Creates a 3x3 matrix

**Explanation:**

* This creates a 2D array with 3 rows and 3 columns.
* All elements are initialized to the default value of int, which is 0.

You can manually assign values like this:

matrix[0][0] = 1; // Set the value at row 0, column 0

matrix[1][1] = 5; // Set the value at row 1, column 1

matrix[2][2] = 9; // Set the value at row 2, column 2

2. Declaration and Direct Initialization:

int[][] matrix = {

{1, 2, 3}, // Row 0

{4, 5, 6}, // Row 1

{7, 8, 9} // Row 2

};

**Explanation:**

* This directly initializes the array with the specified values.
* The structure looks like this

Row 0: [1, 2, 3]

Row 1: [4, 5, 6]

Row 2: [7, 8, 9]

You can access elements using their **row index** and **column index**:

System.out.println(matrix[0][0]); // Output: 1 (Row 0, Column 0)

System.out.println(matrix[1][2]); // Output: 6 (Row 1, Column 2)

**Iterating Over a 2D Array**

**Nested Loops**

To iterate over rows and columns:

for (int i = 0; i < matrix.length; i++) { // Outer loop for rows

for (int j = 0; j < matrix[i].length; j++) { // Inner loop for columns

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

}

System.out.println(); // Move to the next line after each row

}

Output for the matrix above:

1 2 3

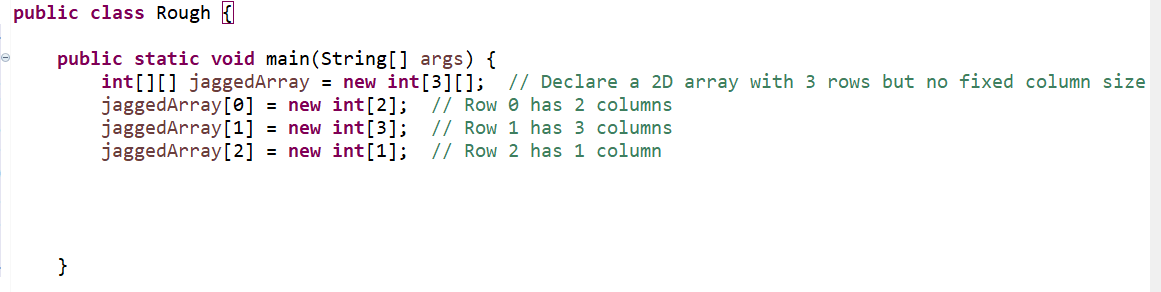
4 5 6

7 8 9

**Jagged Arrays (Irregular 2D Arrays)**

Unlike a standard 2D array, where all rows have the same number of columns, **jagged arrays** allow rows to have different lengths.

**Example:**

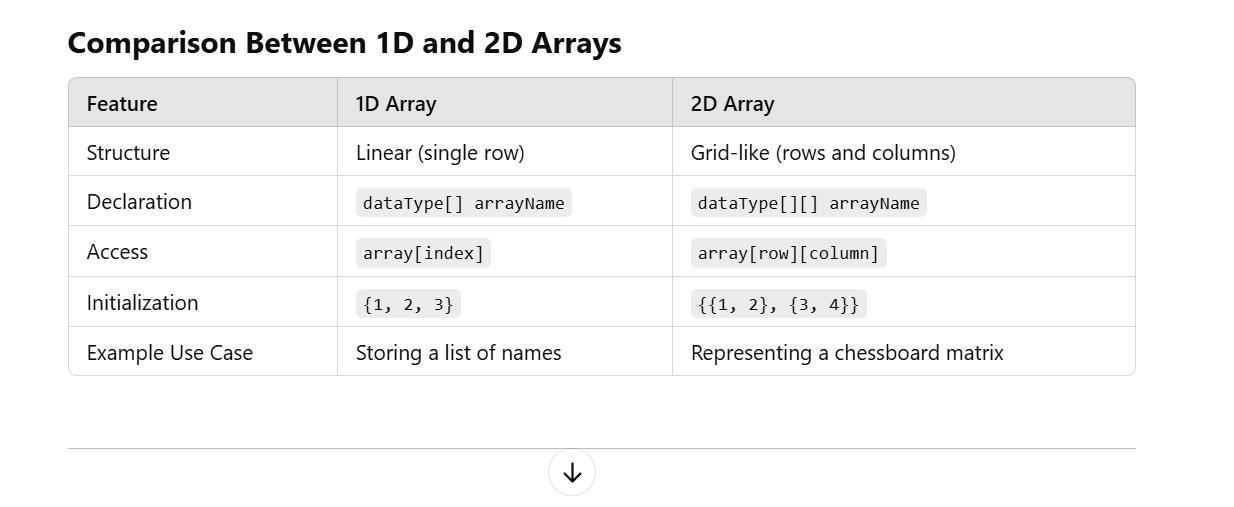


**Structure:**

Row 0: [0, 0]

Row 1: [0, 0, 0]

Row 2: [0]



**Key Notes**

1. Use a **2D array** for data structured in rows and columns (like matrices, tables, or grids).
2. Remember that the first index is the row and the second index is the column in a 2D array.
3. For irregular row sizes, consider using **jagged arrays**.

**2.Difference Between Collection and Collections**

**Collection:**

* **Definition**: It is an **interface** in the java.util package.
* **Purpose**: Represents a group of objects known as elements.
* **Part of**: Java Collections Framework.
* **Key Subinterfaces**:
  + List
  + Set
  + Queue
* **Examples**:
  + ArrayList, HashSet, PriorityQueue implement Collection.

**Collections:**

* **Definition**: It is a **utility class** in the java.util package.
* **Purpose**: Provides static methods to operate on or return collections.
* **Key Methods**:
  + Sorting: Collections.sort(list)
  + Searching: Collections.binarySearch(list, key)
  + Synchronization: Collections.synchronizedList(list)
  + Minimum and Maximum: Collections.min(collection), Collections.max(collection)
* **Not Part of**: The Collection interface.

**2. Overview of the Collection Framework Architecture**

**What is the Collection Framework?**

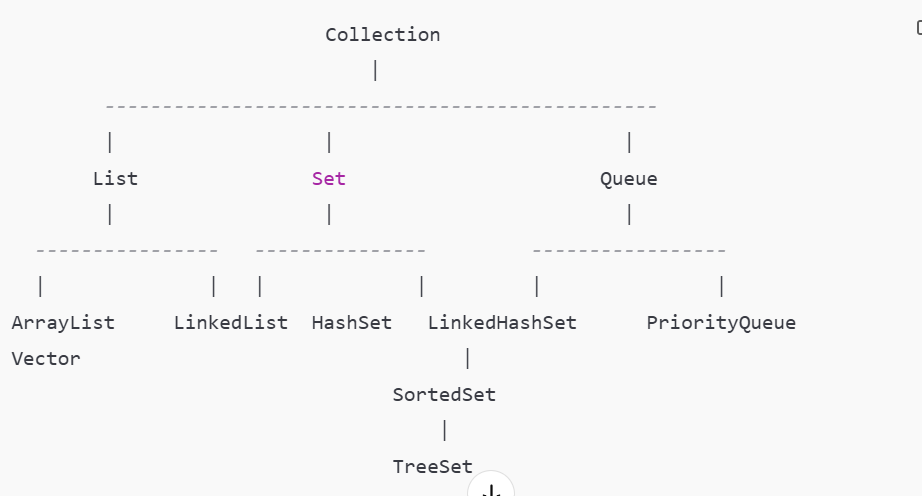
* A **framework** that provides a standard way to organize, manipulate, and store groups of objects in Java.
* **Key Features**:
  + Interfaces: Define the abstract types.
  + Implementations: Provide concrete classes like ArrayList, HashSet.
  + Algorithms: Utilities for sorting, searching, etc.

**Diagram of the Collection Framework**

Here’s the simplified hierarchy:

sql

Copy code



**Key Interfaces in Collection Framework**

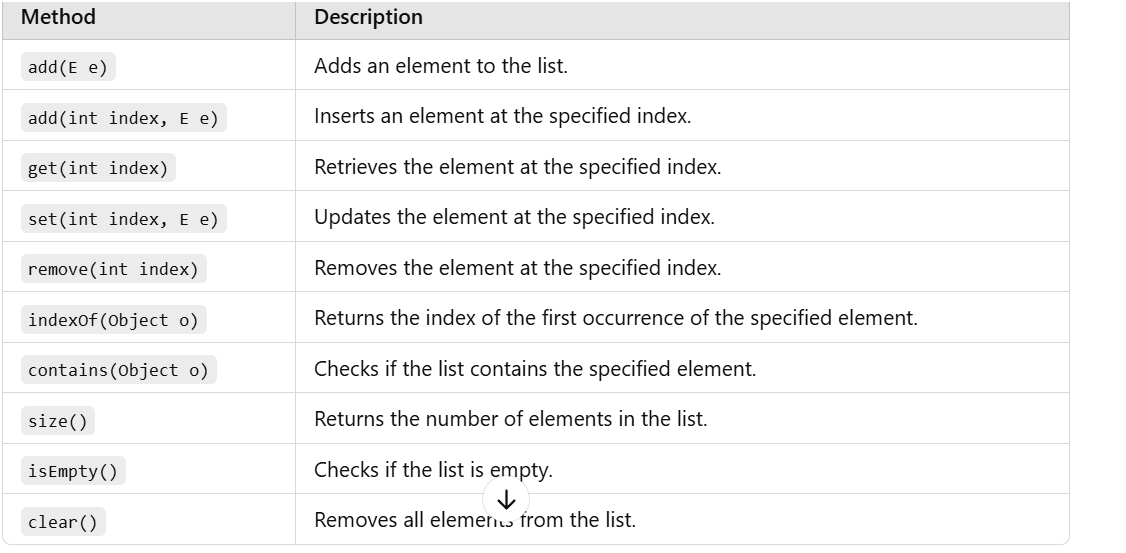
1. **Collection**: Root interface for all collections.
2. **List**: Ordered collection (allows duplicates). Example: ArrayList, LinkedList.
3. **Set**: Unordered collection (no duplicates). Example: HashSet, TreeSet.
4. **Queue**: Collection for holding elements to process in FIFO order. Example: PriorityQueue.
5. **Map**: Maps keys to values (not derived from Collection interface). Example: HashMap.

**2.** **List in Java**

A **List** in Java is an **interface** in the java.util package that is part of the **Java Collections Framework**. It represents an **ordered collection** of elements, where duplicates are allowed, and elements can be accessed using an **index**.

**Key Features of List**

1. **Order**:
   * The elements in a List are stored in the **insertion order**.
   * You can rely on the order in which elements are added to the list.
2. **Duplicates Allowed**:
   * A List can contain duplicate elements.
3. **Index-Based Access**:
   * Elements can be accessed, modified, or removed using their **index** (position in the list).
4. **Dynamic Resizing**:
   * Unlike arrays, most List implementations (e.g., ArrayList) grow dynamically as elements are added.



**List.of() and Immutability**

**What is List.of()?**

* List.of() is a static factory method introduced in **Java 9** to create immutable lists.
* The list created using List.of() is **unmodifiable** and does **not allow null elements**.

**Example**

List<String> words = List.of("a", "b", "c");

System.out.println(words); // Output: [a, b, c]

**Key Characteristics**

1. **Immutability**:
   * Once the list is created, you **cannot modify it** (add, remove, or update elements).

words.add("d"); // Throws UnsupportedOperationException

words.remove("a"); // Throws UnsupportedOperationException

words.set(0, "z"); // Throws UnsupportedOperationException

**No Nulls Allowed**:

* Adding null as an element will throw a NullPointerException.

List<String> invalidList = List.of("a", null); // Throws NullPointerException

**Fixed Size**:

* The size of the list is fixed and cannot be changed.

**List.of() and Immutability**

**What is List.of()?**

* List.of() is a static factory method introduced in **Java 9** to create immutable lists.
* The list created using List.of() is **unmodifiable** and does **not allow null elements**.

**Example**

java

Copy code

List<String> words = List.of("a", "b", "c");

System.out.println(words); // Output: [a, b, c]

**Key Characteristics**

1. **Immutability**:
   * Once the list is created, you **cannot modify it** (add, remove, or update elements).

java

Copy code

words.add("d"); // Throws UnsupportedOperationException

words.remove("a"); // Throws UnsupportedOperationException

words.set(0, "z"); // Throws UnsupportedOperationException

1. **No Nulls Allowed**:
   * Adding null as an element will throw a NullPointerException.

java

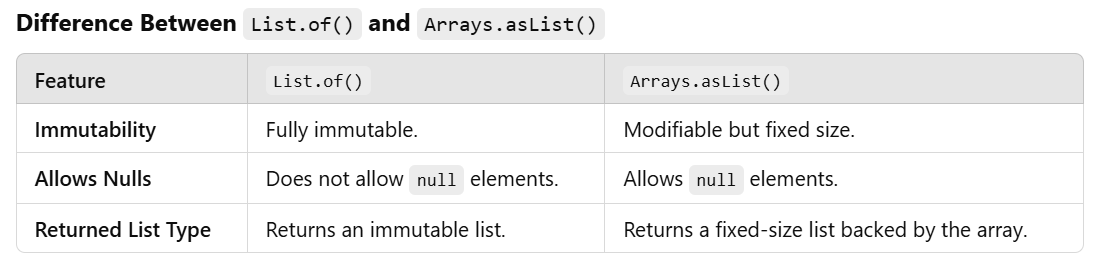
Copy code

List<String> invalidList = List.of("a", null); // Throws NullPointerException

1. **Fixed Size**:
   * The size of the list is fixed and cannot be changed.

**When to Use?**

* Use List.of() when you need a **read-only** list, especially for constants or configuration values.



**3. Detailed Explanation of ArrayList**

**What is ArrayList?**

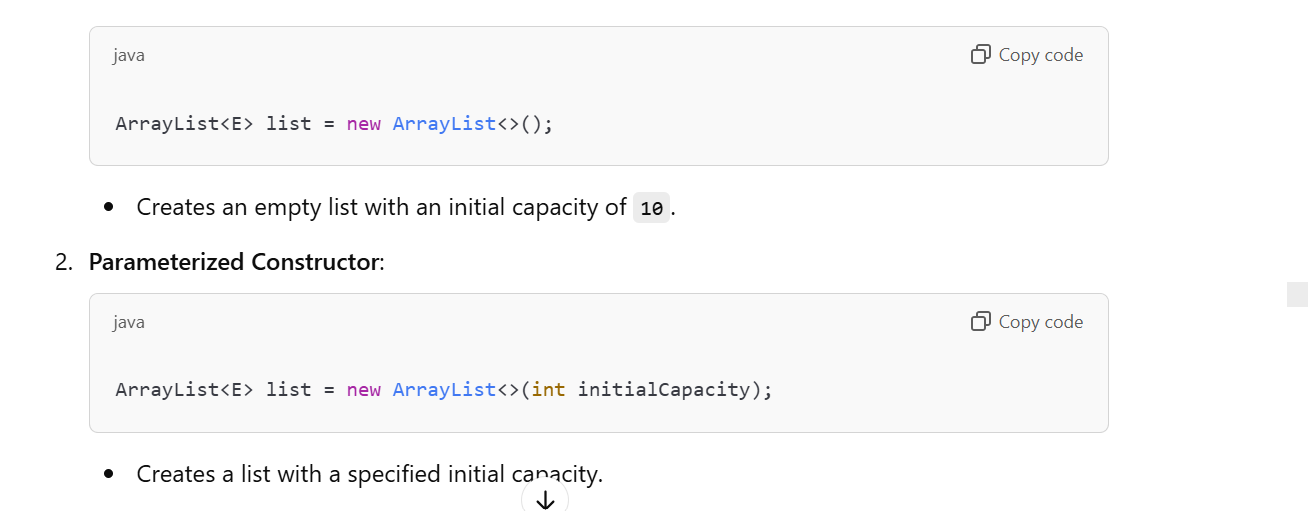
* A **class** in the java.util package.
* Implements the List interface.
* Allows dynamic resizing, unlike arrays (fixed size).
* Stores **elements in insertion order** and allows duplicates.

**Features of ArrayList**

1. **Dynamic Resizing**:
   * Automatically increases its size when elements are added beyond its current capacity.
2. **Indexed Access**:
   * Elements can be accessed using indices, making it faster for read operations.
3. **Allows Null**:
   * null values can be stored.
4. **Non-Synchronized**:
   * Not thread-safe; requires external synchronization for concurrent access.

**Constructors of ArrayList**

1. **Default Constructor**:



1. **Collection Constructor**:



Creates a list containing elements of a specified collection.

**Methods in ArrayList**

1. **Adding Elements**:

list.add("Element"); // Appends at the end

list.add(2, "Indexed"); // Inserts at index 2

1. **Removing Elements**:

list.remove("Element"); // Removes first occurrence

list.remove(1); // Removes element at index 1

1. **Accessing Elements**:

String element = list.get(0); // Access by index

1. **Updating Elements**:

list.set(0, "New Value"); // Update element at index 0

1. **Checking Size**:

int size = list.size(); // Returns the number of elements

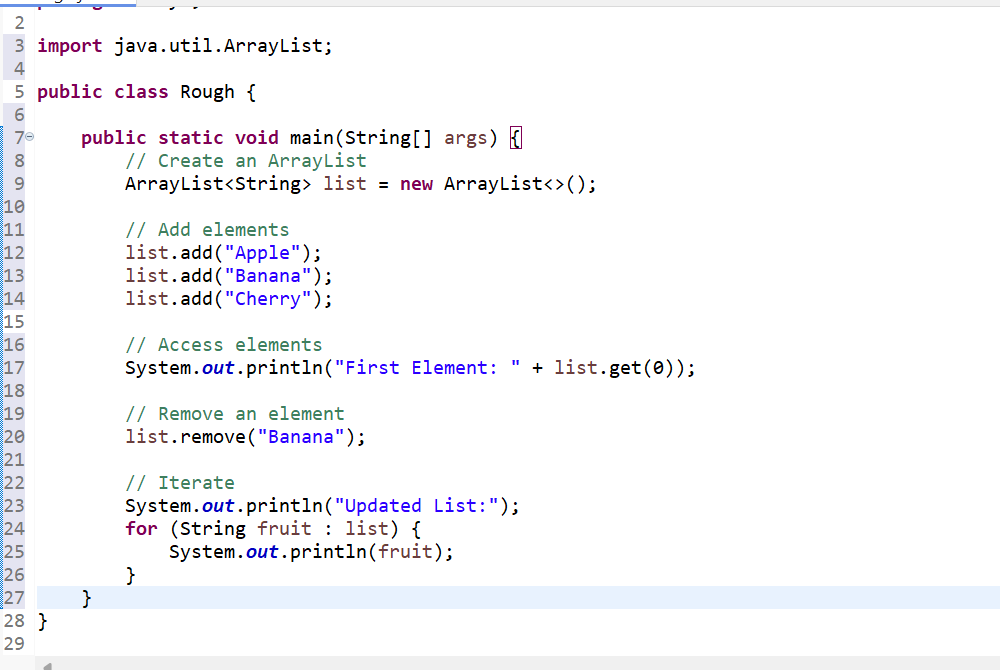
1. **Iterating**:

for (String item : list) {

System.out.println(item);

}

**Example:**



**Advantages of ArrayList**

1. **Dynamic Resizing**: Automatically grows as elements are added.
2. **Indexed Access**: Provides fast random access.
3. **Flexibility**: Allows insertion, deletion, and iteration.

**Disadvantages of ArrayList**

1. **Performance**: Slower compared to arrays for operations like searching.
2. **Not Synchronized**: Requires manual synchronization in multi-threaded environments.

**Interview Questions**

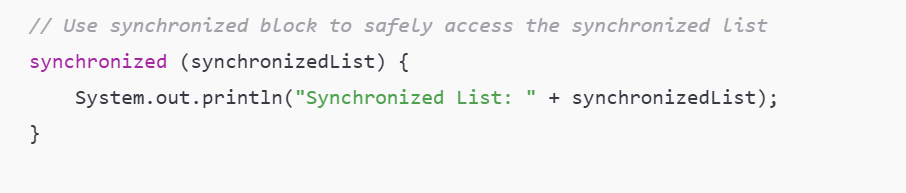
1. **What is the difference between Array and ArrayList?**
   * **Array** is fixed-size; **ArrayList** is dynamic.
   * Array can store primitive data types; ArrayList stores objects.
2. **How does ArrayList grow dynamically?**
   * When the current capacity is exceeded, ArrayList creates a new array with 50% more capacity and copies the old elements to the new array.
3. **Is ArrayList thread-safe?**
   * No, ArrayList is not synchronized. Use Collections.synchronizedList() or CopyOnWriteArrayList for thread safety.
4. **How to synchronize an ArrayList?**

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

**Code Explanation**

1. **Collections.synchronizedList**:
   * Converts a regular ArrayList into a thread-safe list.
   * Synchronization ensures that multiple threads accessing or modifying the list won't cause data corruption.







1. **What is the initial capacity of ArrayList?**

* Default capacity is 10.

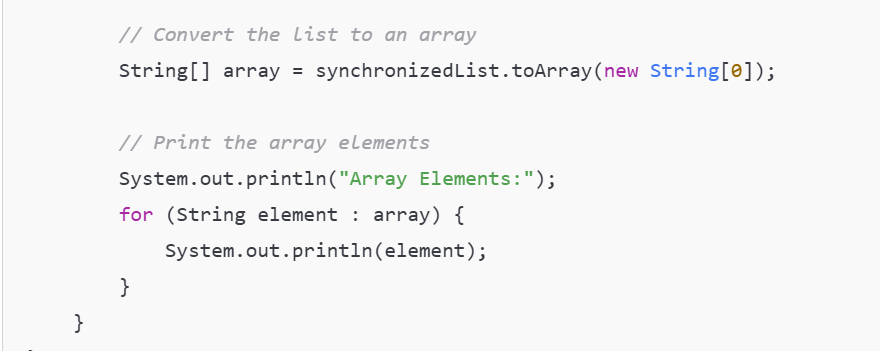
1. **How to convert ArrayList to an array?**

String[] array = list.toArray(new String[0]);

**Code Explanation**

**list.toArray(new String[0])**:

* Converts a List<String> to an array of String objects.
* The new String[0] ensures the correct type of array is returne



0/p



1. **What happens when you add null to an ArrayList?**

* null can be added, and it behaves like any other object.

**Interfaces Implemented by ArrayList**

The ArrayList class implements several interfaces, each contributing to its behavior. Here's a detailed breakdown:

**1. List**

* **Description**:
  + ArrayList implements the List interface, which represents an **ordered collection** of elements.
  + It allows:
    - Duplicate elements.
    - Access by index.
    - Insertion of elements at specific positions.
* **Key Methods from List Interface**:
  + add(E e): Adds an element to the list.
  + add(int index, E element): Inserts an element at a specific position.
  + get(int index): Returns the element at the specified index.
  + remove(int index): Removes the element at the specified index.

**2. RandomAccess**

* **Description**:
  + This is a **marker interface** used to indicate that the list supports fast (constant-time) random access.
  + ArrayList implements this because it allows direct access to elements using their index.
* **Purpose**:
  + Enhances performance for scenarios where random access (e.g., get(index)) is frequently used.

**3. Cloneable**

* **Description**:
  + Indicates that the class supports cloning, meaning it can create a **shallow copy** of the ArrayList.
  + The clone() method creates a new ArrayList object with the same elements.
* **Usage**:

java

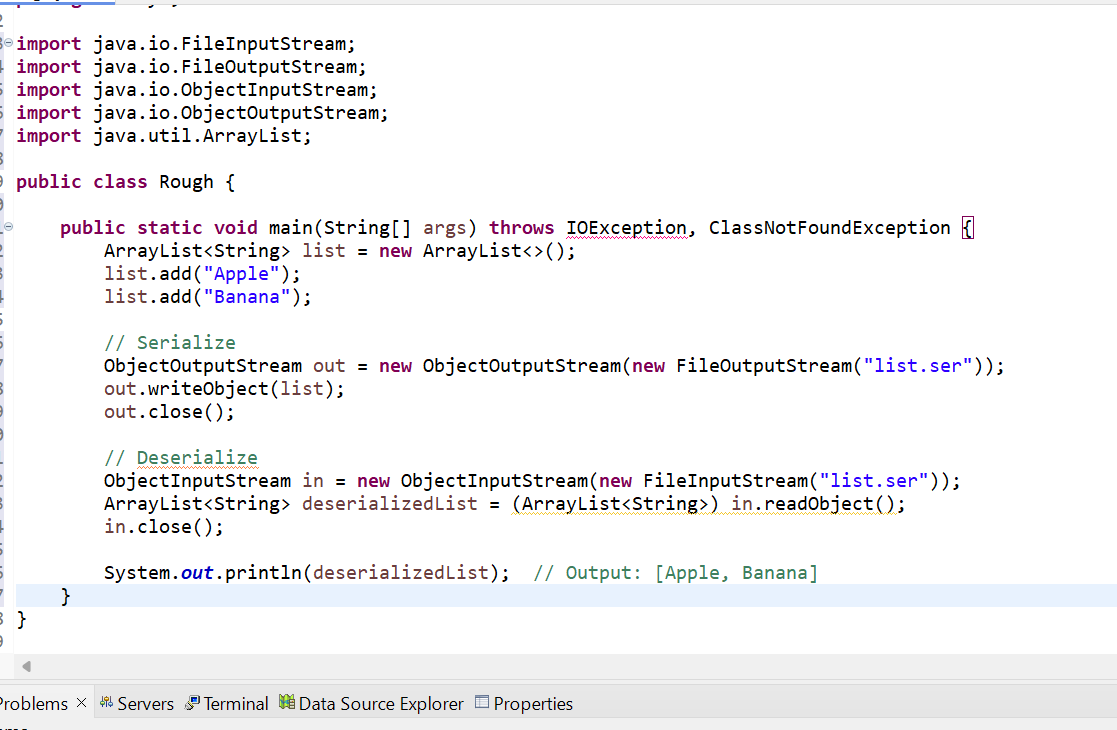
Copy code



**Serializable**

* **Description**:
  + Enables ArrayList objects to be serialized, meaning their state can be saved to a file or transferred over a network.
  + Required for persistence or communication purposes.
* **Usage**:
  + Serialization can be done using ObjectOutputStream and deserialization using ObjectInputStream.
  + Example:

java



4.**LinkedList in Java**

The LinkedList class in Java is part of the java.util package and implements the **List**, **Deque**, and **Queue** interfaces. It represents a doubly linked list, where each node contains:

1. A reference to the previous node.
2. A reference to the next node.
3. The data.

**Key Characteristics of LinkedList**

1. **Dynamic Size**:
   * The size of a LinkedList can dynamically grow or shrink during runtime.
2. **Efficient Insertions/Deletions**:
   * Inserting or removing elements from the middle or ends of a LinkedList is faster compared to an ArrayList because it doesn’t involve shifting elements.
3. **Non-Contiguous Storage**:
   * Elements are stored in **nodes** that are linked using pointers/references, unlike ArrayList where elements are stored in contiguous memory.

In **ArrayList**, elements are stored in **contiguous memory** (one after another in a single block). For example, an ArrayList of size 5 would look like this in memory:

[Element1] [Element2] [Element3] [Element4] [Element5]

* In **LinkedList**, elements are stored in **nodes**, and each node contains:
  1. **Data**: The actual value stored.
  2. **Pointers/References**:
     + One pointer to the previous node.
     + One pointer to the next node.
  3. The nodes are scattered in memory but linked together using these pointers.

**Structure of LinkedList**

For example, a LinkedList storing "A", "B", "C" might look like this in memory:

[Null <- A -> Address1] -> [Address0 <- B -> Address2] -> [Address1 <- C -> Null]

Here:

* Each node points to the previous node (<-) and the next node (->).
* Nodes are **not contiguous** (not stored next to each other).
* **Contiguous memory** means all elements are stored in a continuous block of memory locations.
* If the array runs out of space, a new array with a larger size is created, and all elements are copied to the new array (which is expensive).

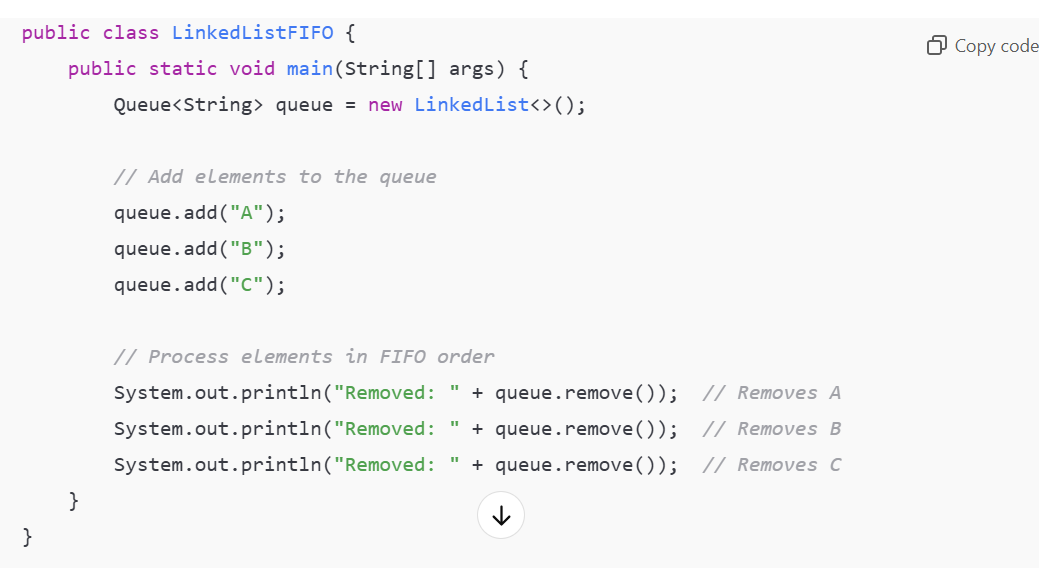
1. **Implements Deque and Queue**:
   * Can be used as a **FIFO (First In, First Out)** or **LIFO (Last In, First Out)** queue.

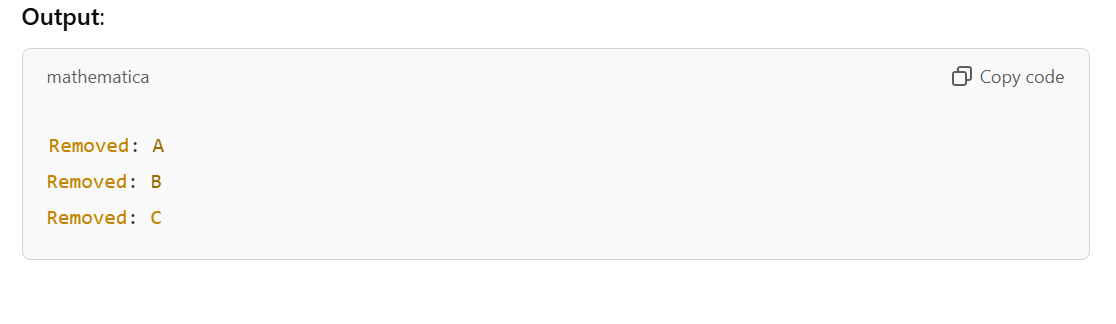
**What is Deque?**

* **Deque** stands for **Double-Ended Queue**.
* It allows adding and removing elements from **both ends** of the list.
* A LinkedList implements the Deque interface, so it can function as:
  + **FIFO (First In, First Out)**: Like a **Queue**.
  + **LIFO (Last In, First Out)**: Like a **Stack**.

**LinkedList as a Queue (FIFO)**

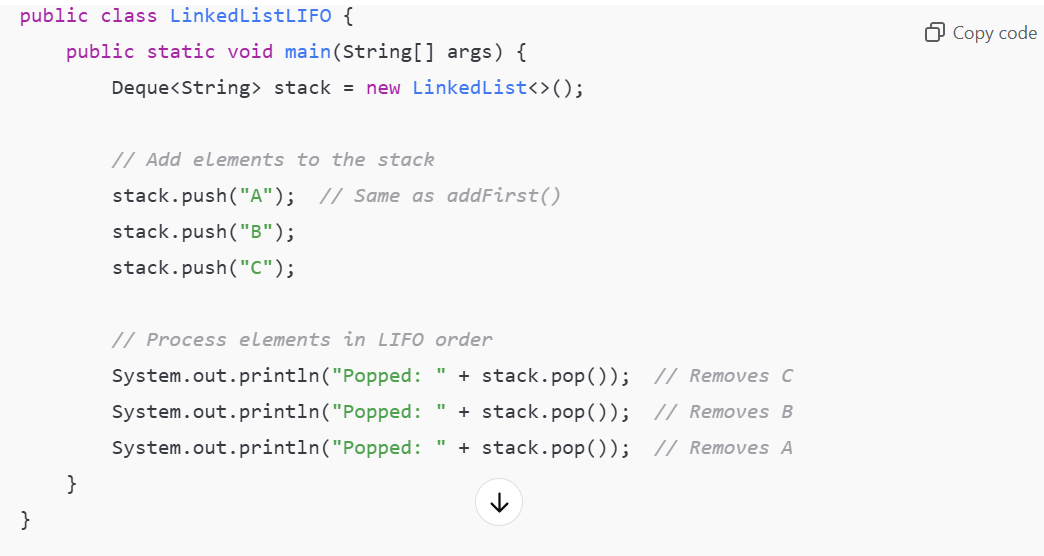
* **FIFO (First In, First Out)** means elements are processed in the order they are added (like a line at a ticket counter).
* LinkedList provides methods to act as a queue:
  + addLast(E e): Add elements at the **end** of the queue.
  + removeFirst(): Remove elements from the **start** of the queue.

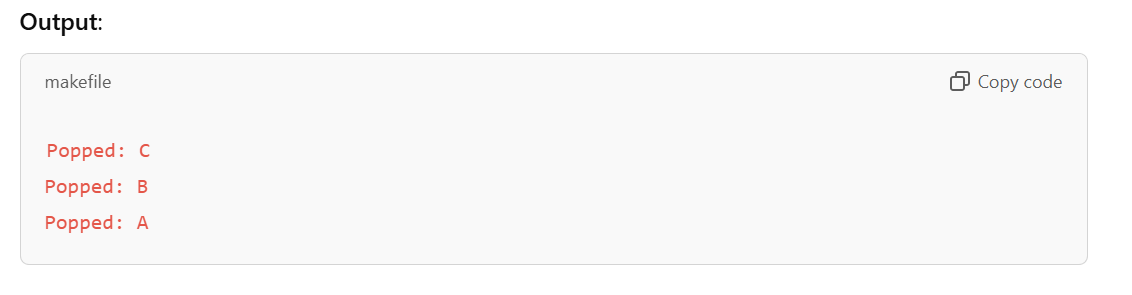




**LinkedList as a Stack (LIFO)**

* **LIFO (Last In, First Out)** means the last element added is the first to be removed (like stacking plates).
* LinkedList provides methods to act as a stack:
  + addFirst(E e): Add elements at the **beginning** of the stack.
  + removeFirst(): Remove elements from the **beginning** of the stack.





**Constructors**

1. **Default Constructor**:

LinkedList<E> list = new LinkedList<>();

Creates an empty linked list.

1. **Collection Constructor**:

LinkedList<E> list = new LinkedList<>(Collection<? extends E> c);

Creates a linked list containing the elements of the given collection.

**Key Methods of LinkedList**

**Basic Operations**

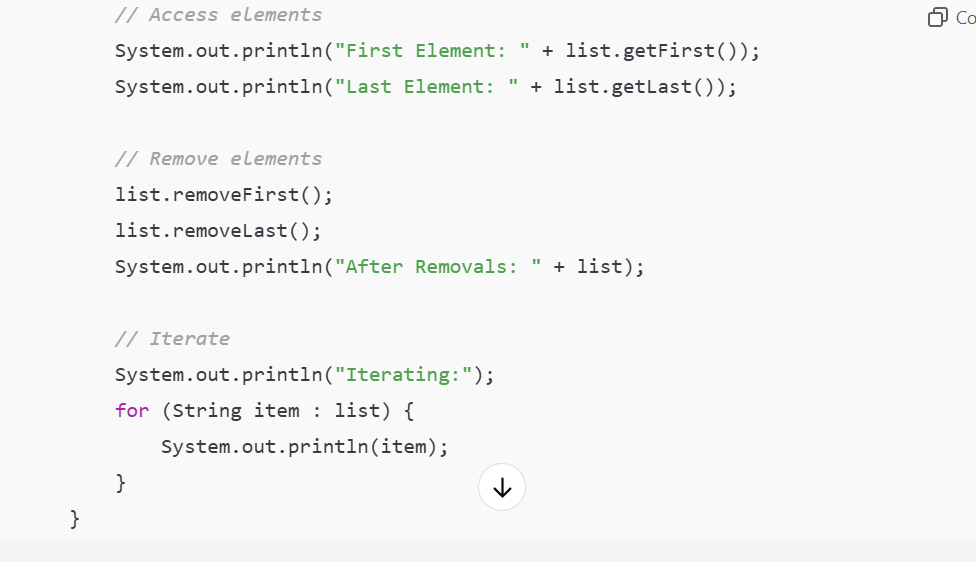
1. **Add Elements**:

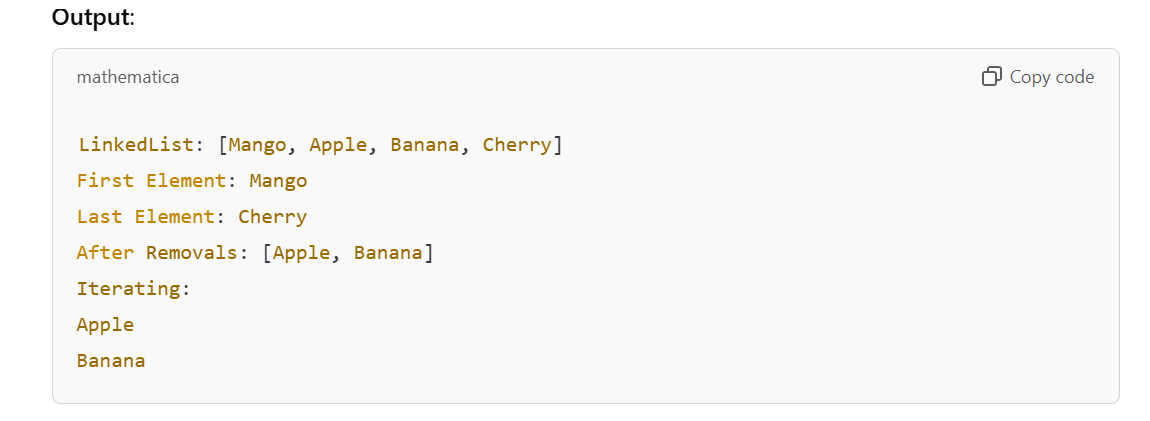




**Examples**





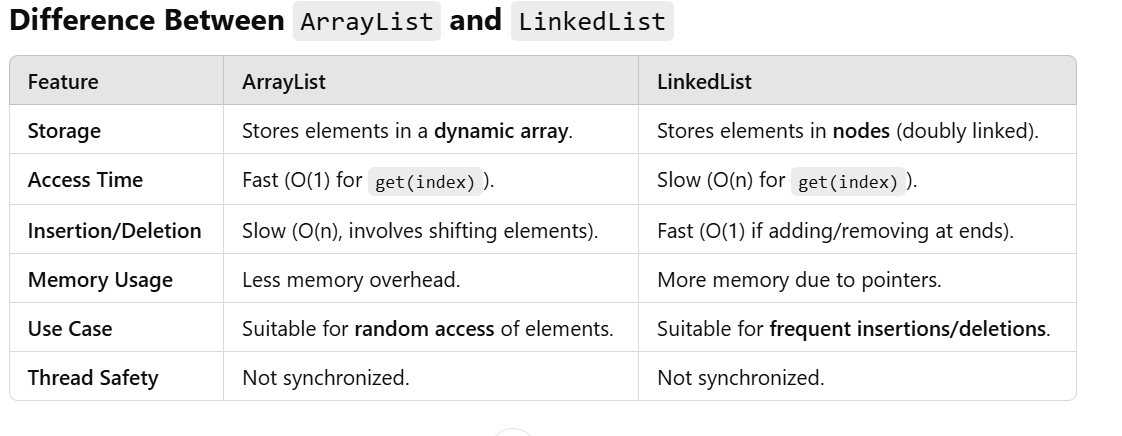


**Advantages of LinkedList**

1. **Efficient Insertions and Deletions**:
   * Inserting or deleting elements is faster as no shifting is required.
2. **Flexible Storage**:
   * Can grow or shrink dynamically without resizing overhead.
3. **Efficient for Queue Operations**:
   * Adding/removing from the beginning or end is efficient.

**Disadvantages of LinkedList**

1. **Slow Access Time**:
   * Accessing an element by index (get(index)) takes **O(n)** time because traversal is required.
2. **Memory Overhead**:
   * Each node requires extra memory for pointers to the previous and next nodes.
3. **Not Cache-Friendly**:
   * Elements are not stored in contiguous memory, making it slower compared to ArrayList for traversal.



**When to Use LinkedList?**

* When frequent insertions or deletions are required in the middle of a collection.
* When you need a **FIFO (Queue)** or **LIFO (Deque/Stack)** data structure.
* When memory allocation is not a concern.

**When to Avoid LinkedList?**

* When frequent random access (via indices) is required.
* When memory usage is a constraint.
* For small datasets where the overhead of pointers is unnecessary.

**Interview Questions on LinkedList**

1. **What is the difference between ArrayList and LinkedList?**
   * Refer to the comparison table above.
2. **What is the time complexity of operations in LinkedList?**
   * Insertion/Deletion (at the ends): **O(1)**.
   * Insertion/Deletion (in the middle): **O(n)**.
   * Access (by index): **O(n)**.
3. **Can a LinkedList contain null values?**
   * Yes, LinkedList allows null values.
4. **Why is LinkedList not cache-friendly?**
   * Elements are scattered in memory, making it less efficient for traversals.
5. **Does LinkedList allow duplicate elements?**
   * Yes, it allows duplicates like other List implementations.
6. **How does a LinkedList grow dynamically?**
   * Nodes are created dynamically as elements are added, without resizing.
7. **How can you synchronize a LinkedList?**

List<String> synchronizedLinkedList = Collections.synchronizedList(new LinkedList<>());

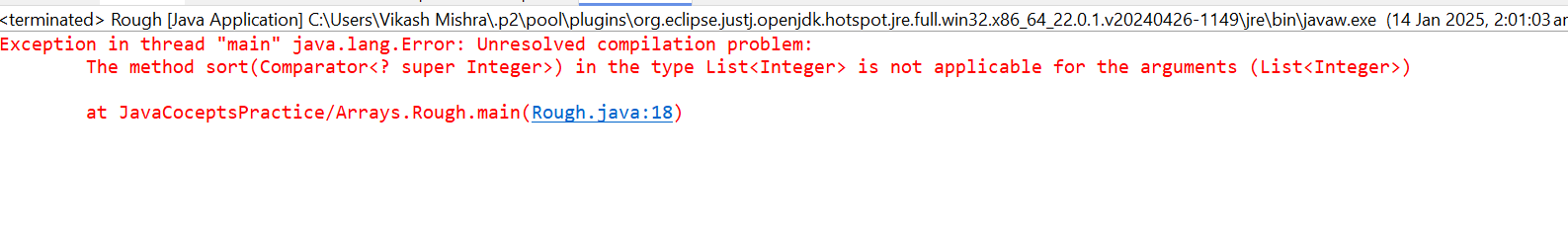
1. **Which is better for implementing a stack or queue: ArrayList or LinkedList?**

* LinkedList is better for stacks (LIFO) and queues (FIFO) because of its efficient addition/removal at ends.

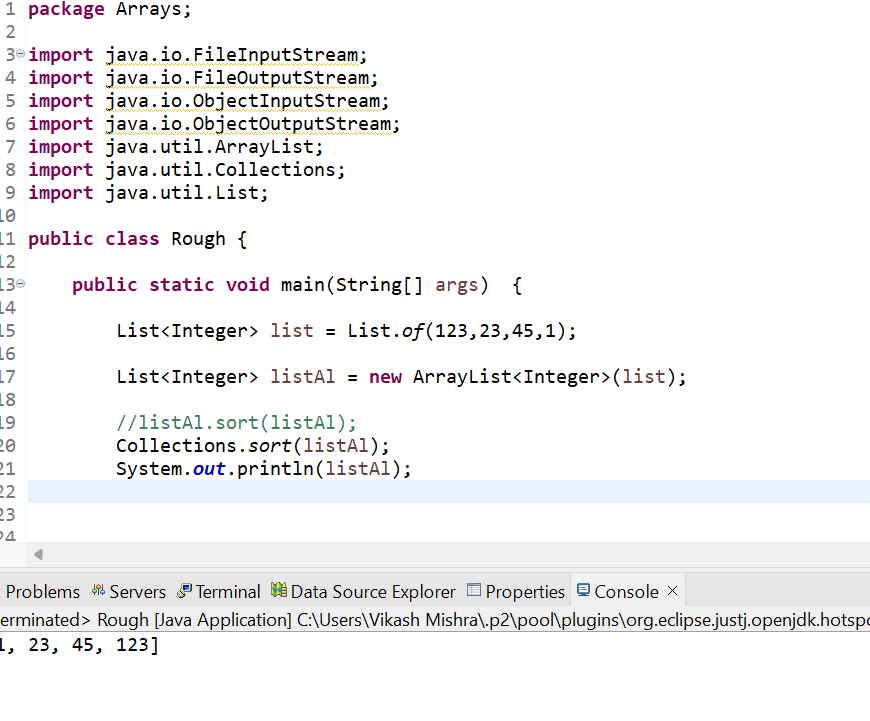
**Sorting With List and Array List**



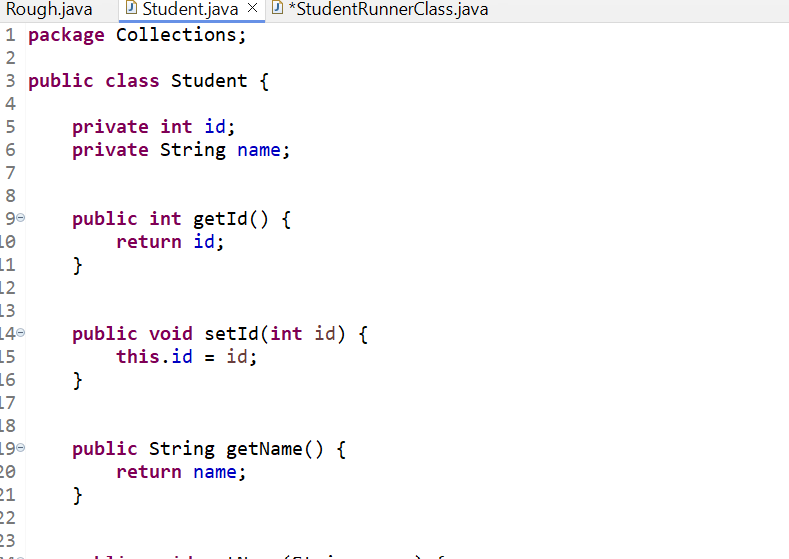
It will throw exception

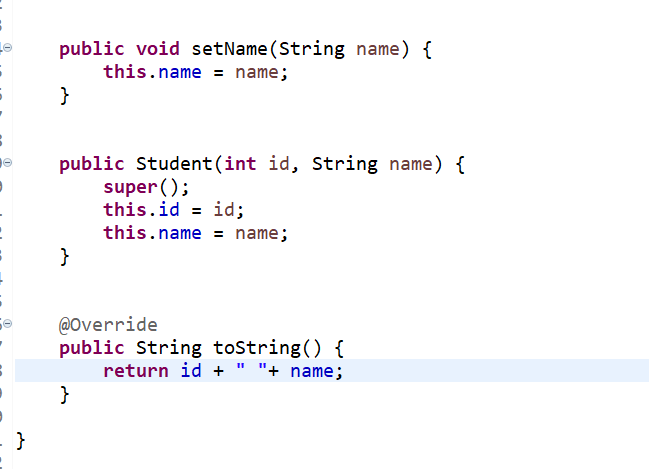


So we will sort through Collection.sort

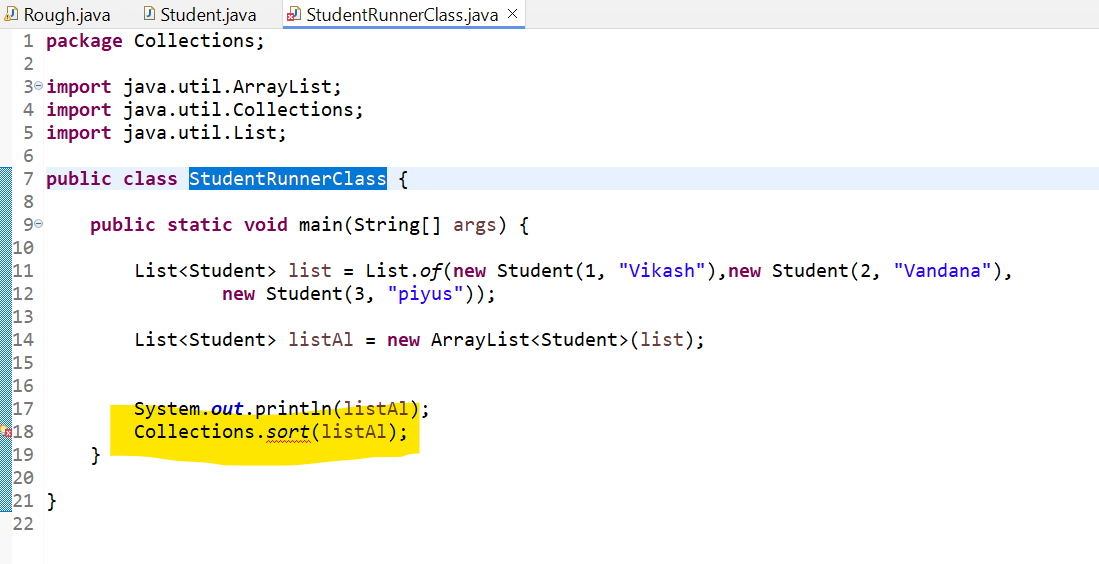


Let’s Play with the sorting with advance concepts

Create student class  




Create StudentRunnerClass



You can see there is compilation error

The issue lies in the Collections.sort(listAl) statement. You cannot sort the listAl directly because the Student class does not implement the Comparable interface or you haven't provided a custom Comparator to the Collections.sort() method.

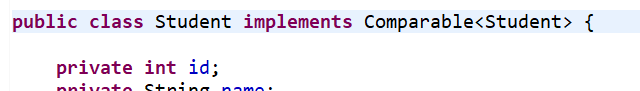
**Explanation**

The Collections.sort() method sorts the elements of a list based on their **natural order**. For this to work:

1. The class (here Student) must implement the Comparable interface and define the **natural order** by overriding the compareTo() method.
2. Alternatively, you must pass a custom Comparator to the Collections.sort() method to define the sorting logic.

**First solution**

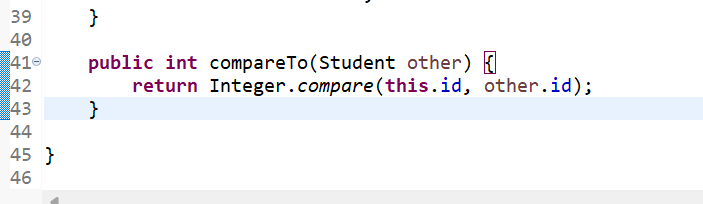
Add the below method in student class



And add this method

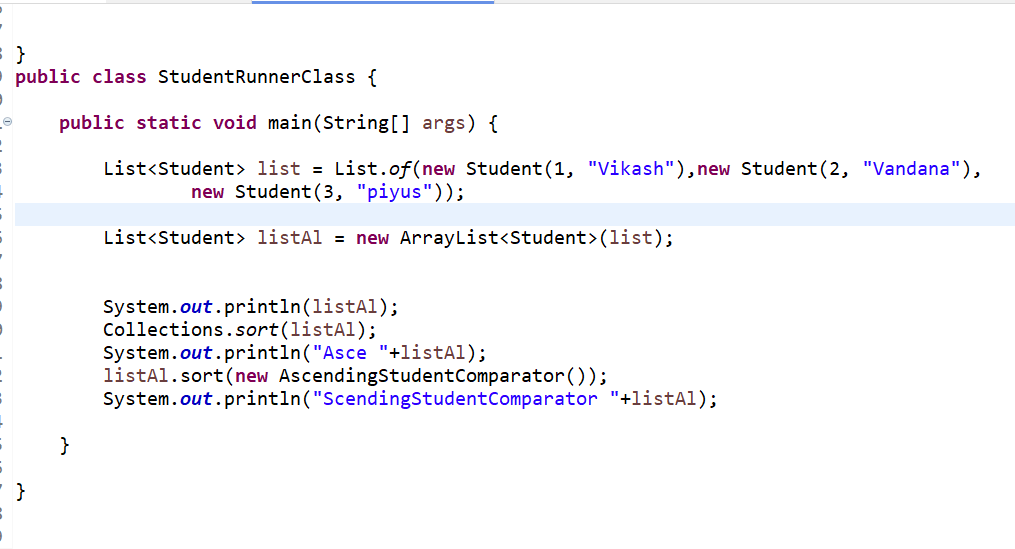
**Explanation**

1. **implements Comparable<Student>**:
   * The Comparable interface defines the natural ordering of objects.
   * The compareTo method is used by Collections.sort() to compare and order elements.
2. **Method Signature of compareTo**:

****

* + Compares the id of two Student objects.
  + Returns:
    - 0 if both are equal.
    - A negative number if this.id is less than other.id.
    - A positive number if this.id is greater than other.id.

Another way to do it



**Explanation of Sorting Using Comparator**

In this code, sorting is achieved using a custom comparator (AscendingStudentComparator) instead of relying on the natural ordering (Comparable).

**Key Points About Comparator**

1. **Comparator Interface**:
   * Provides a way to define custom sorting logic.
   * You do not need to modify the Student class or make it implement Comparable.
2. **Custom Comparator**:
   * A class implementing the Comparator<Student> interface can provide specific sorting logic by overriding the compare method.

**Using the Comparator for Sorting**:

* listAl.sort(new AscendingStudentComparator());
  + This sorts the listAl based on the custom logic defined in AscendingStudentComparator.

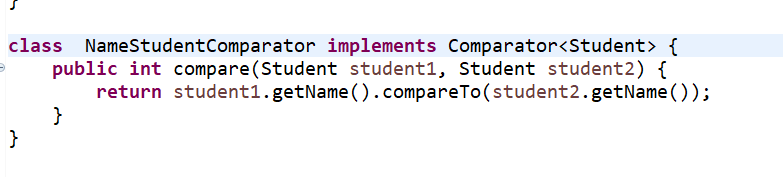
Alternatively, you can use Collections.sort with the comparator:

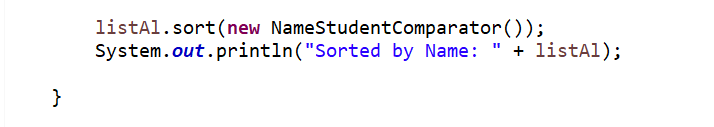
Collections.sort(listAl, new AscendingStudentComparator());

**Adding a Comparator for Sorting by Name**

To sort by name, you can create another custom comparator that compares Student objects based on their name.

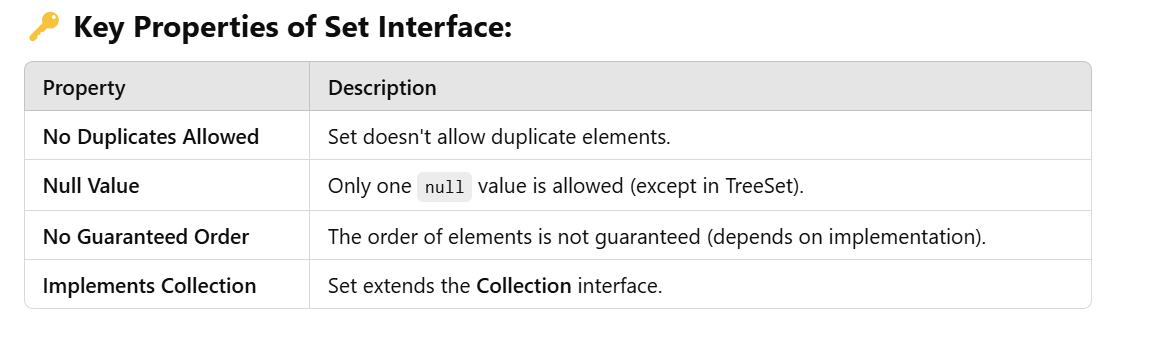
**Name-Based Comparator**





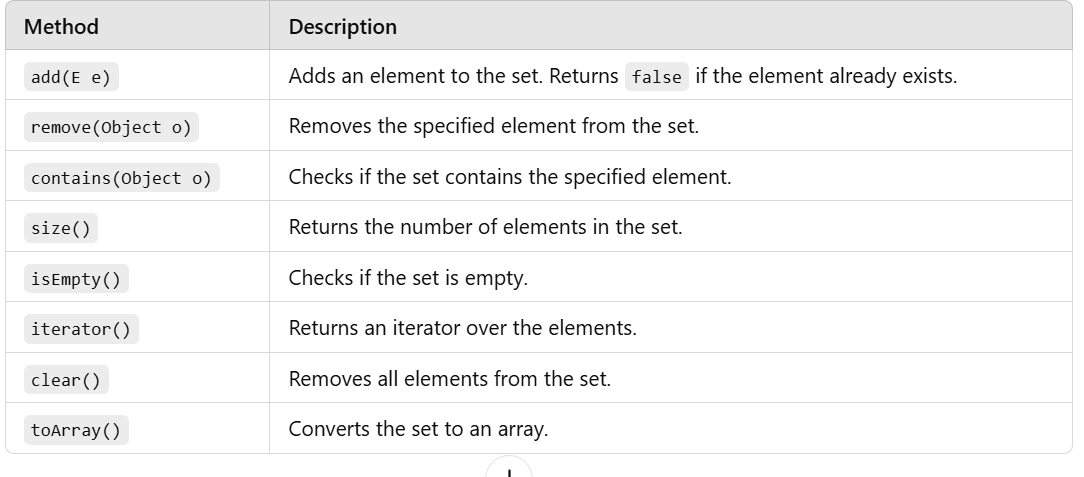
**5.What is the Set Interface?**

The **Set interface** is part of the **Java Collection framework**. It represents a **collection of unique elements** where **no duplicate values** are allowed.



**Methods in the Set Interface:**

Since **Set** extends the **Collection** interface, it inherits all its methods. Here are the key methods of **Set**:



**Classes that Implement the Set Interface:**

1️⃣ **HashSet**

* **Uses HashMap internally** to store elements.
* **Does not maintain order** of elements.
* Allows **one null value**.
* **Fast** for operations like add, remove, and search.

2️⃣ **LinkedHashSet**

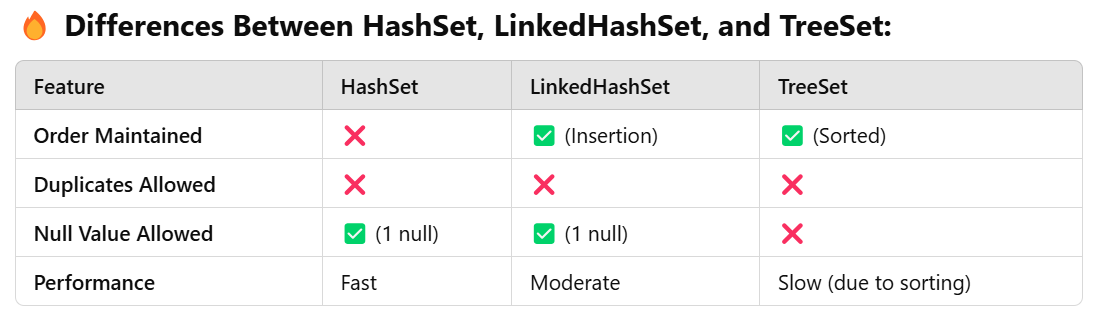
* Inherits from **HashSet**.
* **Maintains insertion order**.
* Allows **one null value**.

3️⃣ **TreeSet**

* Implements **NavigableSet** and **SortedSet**.
* Based on **Red-Black Tree**.
* **Maintains elements in sorted order**.
* **Does not allow null values**.

**📝 Common Use Cases of Set:**

* Removing duplicate values from a list.
* Storing unique elements like **IDs, usernames, or emails**.
* Performing **set operations** like union, intersection, and difference.



**Understanding the Data structure of Array, LinkedList and hashing**

**1. Array Data Structure**

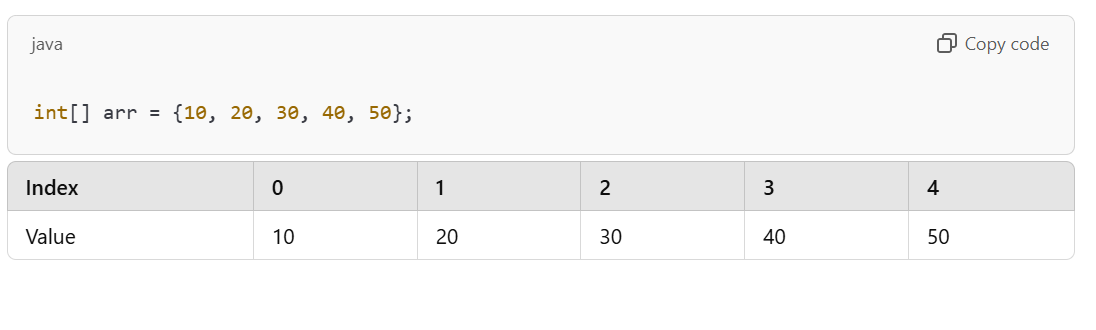
**🔎 What is an Array?**

An **array** is a **linear data structure** that stores elements in **contiguous memory locations**. Each element in an array is accessed using an **index**.

**📂 How Array Stores Data:**

* The **size** of the array is fixed when it's created.
* All elements are stored **sequentially** in memory.
* Elements are accessed using an **index starting from 0**.

**📊 Example:**



**Key Points:**

* **Access Time**: O(1) (because of indexing)
* **Insertion/Deletion Time**: O(n) (because elements need to shift)
* **Fixed Size**: Once created, the size of the array cannot be changed.

**🔧 Operations in Array:**

1. **Access**: Access any element using its index (e.g., arr[2] gives 30).
2. **Insert**: Inserting in the middle requires shifting elements.
3. **Delete**: Deleting requires shifting elements to fill the gap.

**2. LinkedList Data Structure**

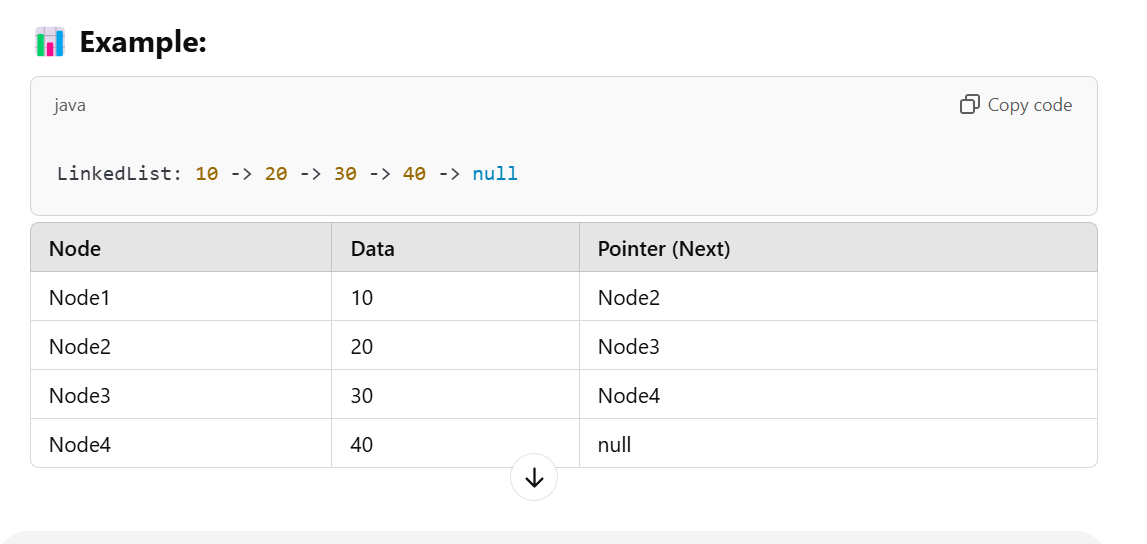
**🔎 What is a LinkedList?**

A **LinkedList** is a **linear data structure** where **each element (node)** contains:

1. **Data**: The value of the node.
2. **Pointer**: A reference to the **next node** in the list.

**📂 How LinkedList Stores Data:**

* The elements are **not stored in contiguous memory locations**.
* Each node points to the **next node** in the list.
* The **last node points to null**, indicating the end of the list.



**Key Points:**

* **Dynamic Size**: The size of the LinkedList can grow or shrink as needed.
* **Insertion/Deletion**: Easier compared to arrays because no shifting is required.
* **Access Time**: O(n) (because you need to traverse the list to find an element).

**🔧 Operations in LinkedList:**

1. **Add**: Add a new node at the end or beginning.
2. **Remove**: Remove a node by updating the pointers.
3. **Traverse**: Go through each node to find or update data.

**Real-Life Example (Train Coaches)**

Imagine you are adding a new coach **(25)** between two existing coaches **(20)** and **(30)**:

1. **Unhook the chain** between **20** and **30**.
2. **Hook the new coach (25)** to **Coach 20**.
3. **Connect the chain from 25 to 30**

**Hashing Concept**

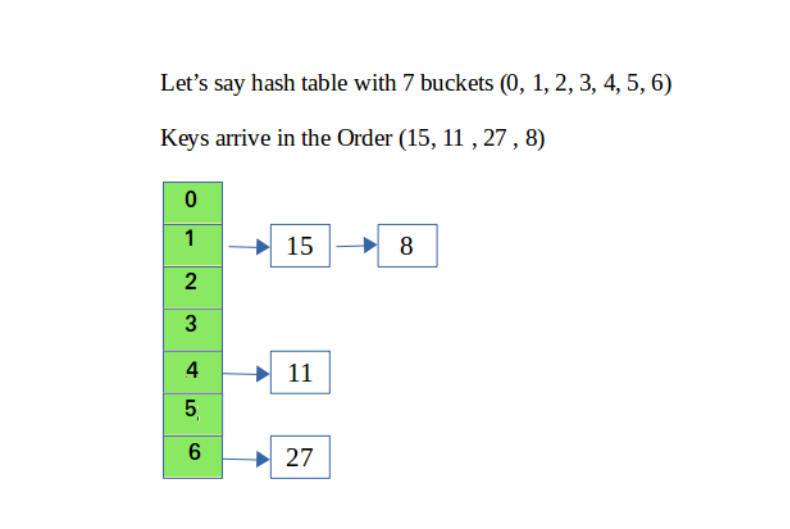
**What is Hashing?**

Think of **Hashing** as a way to **store and find data quickly** using a **key-value pair**.

Instead of searching through all the data (like in an array or LinkedList), **hashing** uses a **hash function** to calculate an **index** where the data should be stored.

**How HashMap Stores Data:**

1. A **HashMap** has an **array of buckets**.
2. Each **bucket** is like a slot in the array where values are stored.
3. The **hash function** takes the **key** and calculates the **index** of the bucket



**What is a Hash Table?**

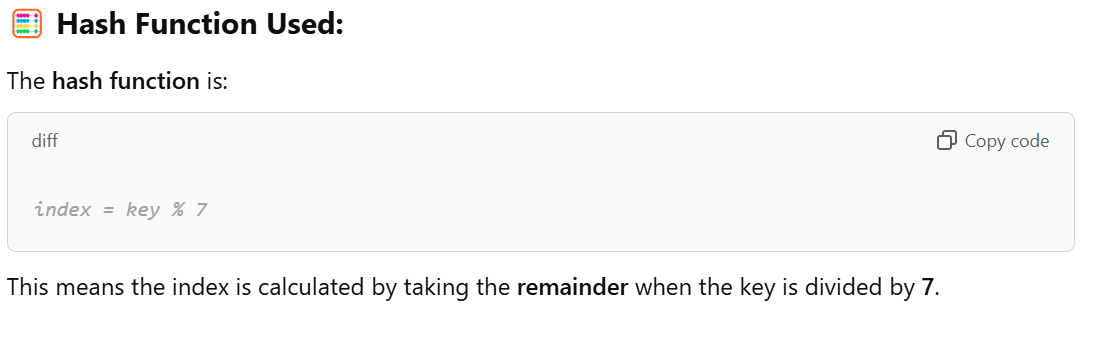
A **hash table** is a **data structure** that stores key-value pairs.  
The key is passed through a **hash function** to determine the **index** (bucket) in the array where the value should be stored.

**📚 Given Hash Table Example:**

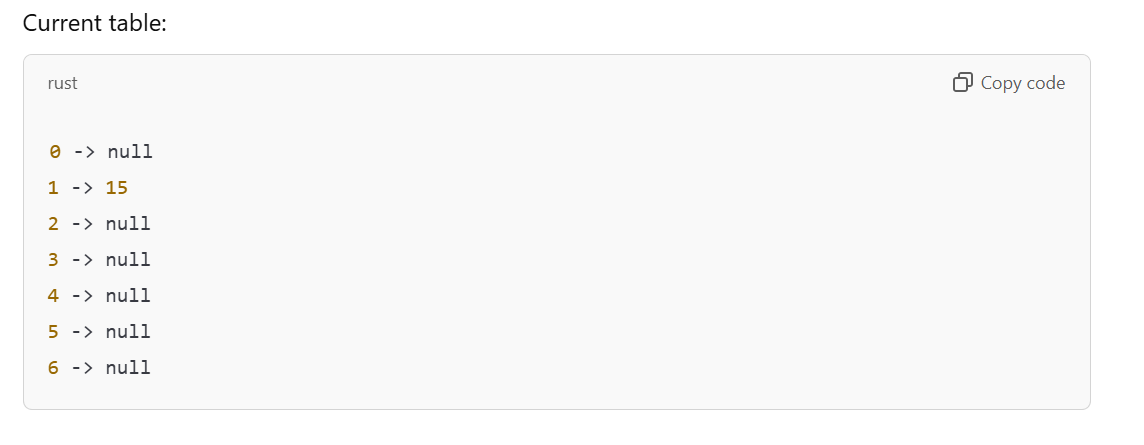
* The hash table has **7 buckets** (index 0 to 6).
* Keys arrive in the order: **15, 11, 27, 8**.
* We use the **modulus operator (%)** as the hash function.

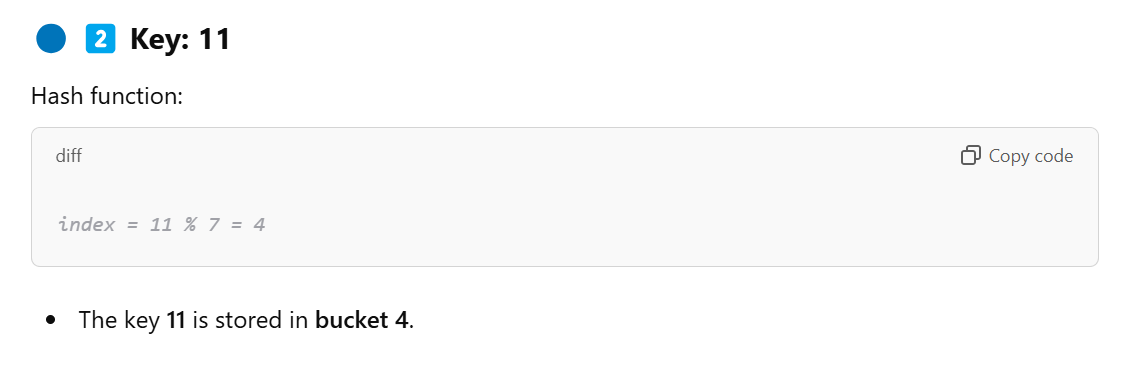
**🧮 Hash Function Used:**

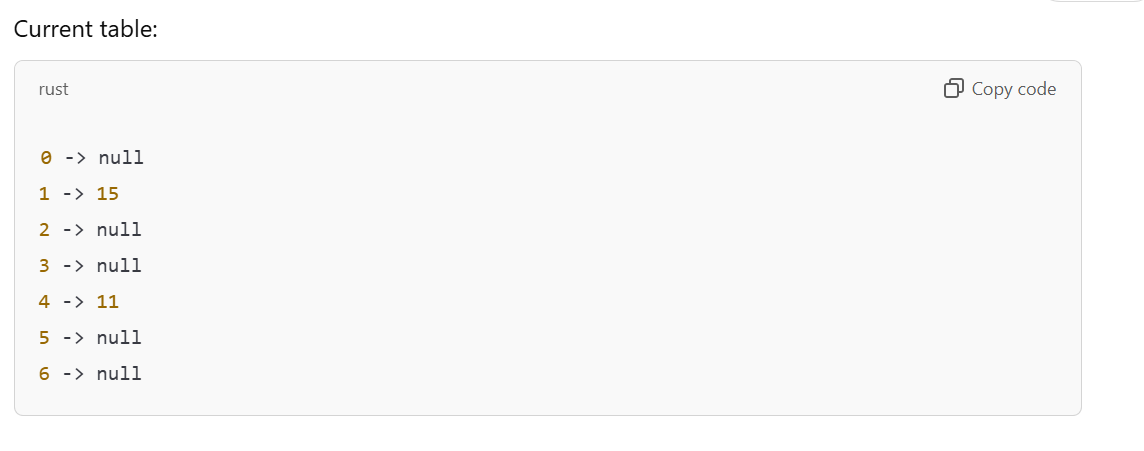
The **hash function** is:

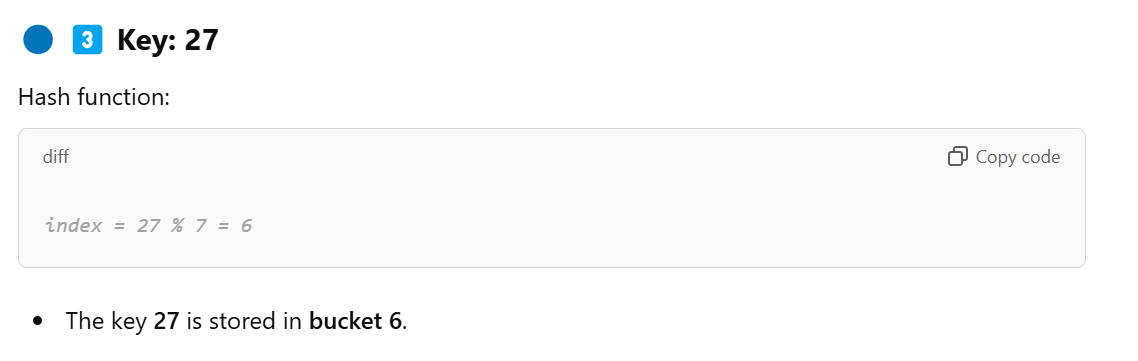


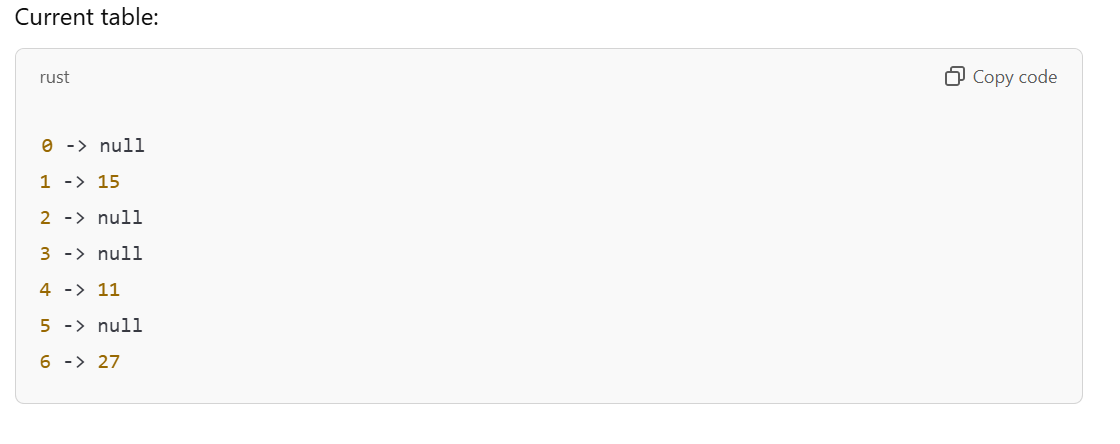


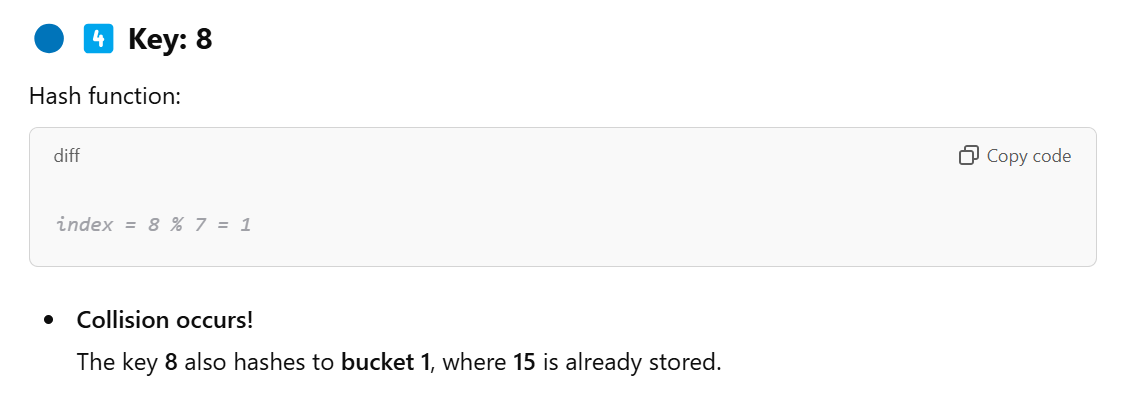








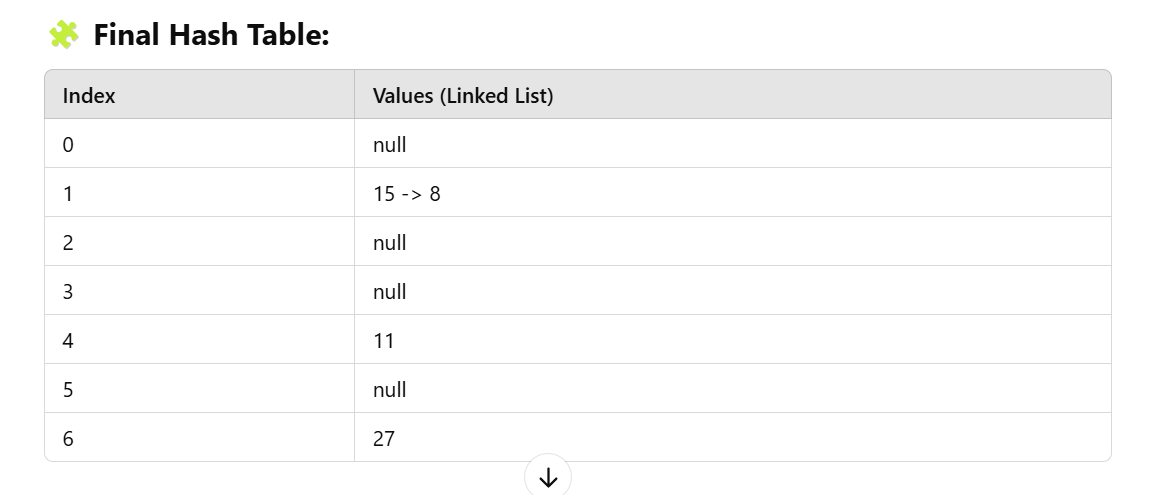


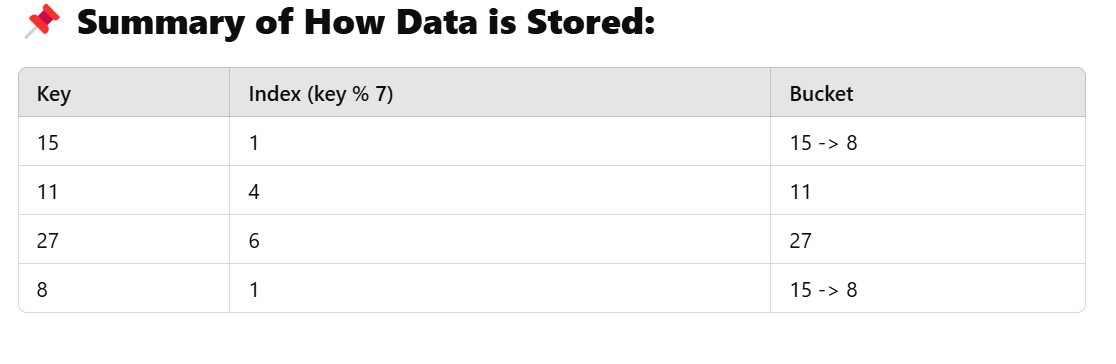


**Solution: Collision Handling Using Chaining**

In the hash table, collisions are handled using **chaining**. This means that each bucket holds a **linked list** to store multiple keys.

* **8** is added to the **linked list** at **bucket 1**, after **15**.





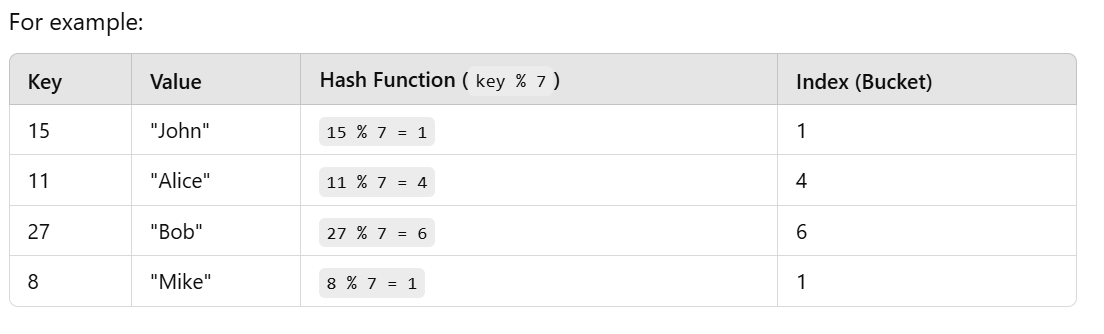
**How Hash Table Stores Keys and Values**

In a **HashMap (or hash table)**, we store data in **key-value pairs**. Here’s how the process works internally.

**✅ Step 1: Applying the Hash Function**

The **key** is passed to a **hash function**, which determines the **index (bucket)** where the key-value pair will be stored.

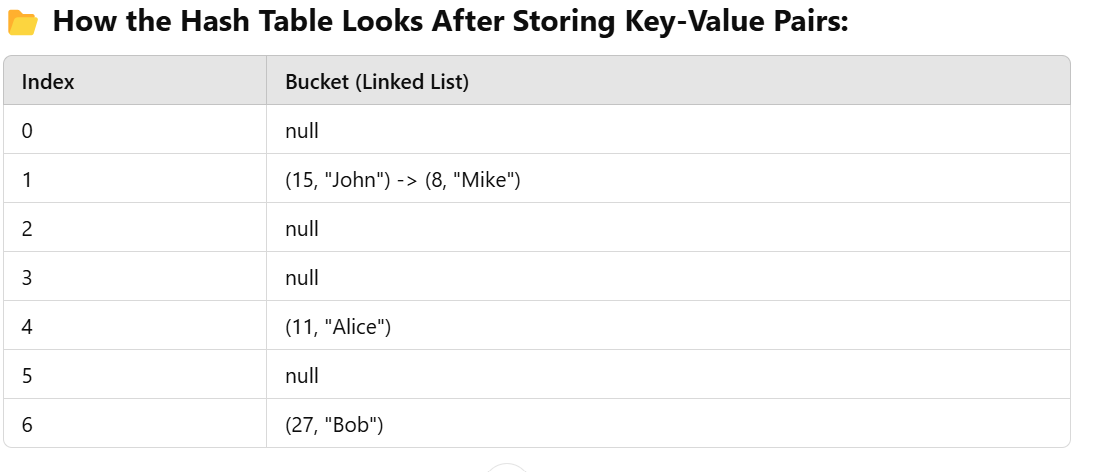
For example:



**tep 2: Storing Key-Value Pairs in the Hash Table**

Each **bucket (index)** in the hash table stores a **LinkedList of nodes**.  
Each **node** contains:

1. **Key** – The unique identifier (like 15, 11, 27).
2. **Value** – The data associated with that key (like "John", "Alice").
3. **Pointer** – Points to the next node in case of a collision (linked list chaining).



**Step 3: What Happens When a Collision Occurs?**

When two keys hash to the **same index**, a **collision** occurs.

In your example:

* **Key 15** hashes to **index 1**.
* **Key 8** also hashes to **index 1**.

To handle the collision, **HashMap uses chaining**:

* At **index 1**, the bucket now contains a **LinkedList**.
* The first node stores **(15, "John")**.
* The second node stores **(8, "Mike")**.

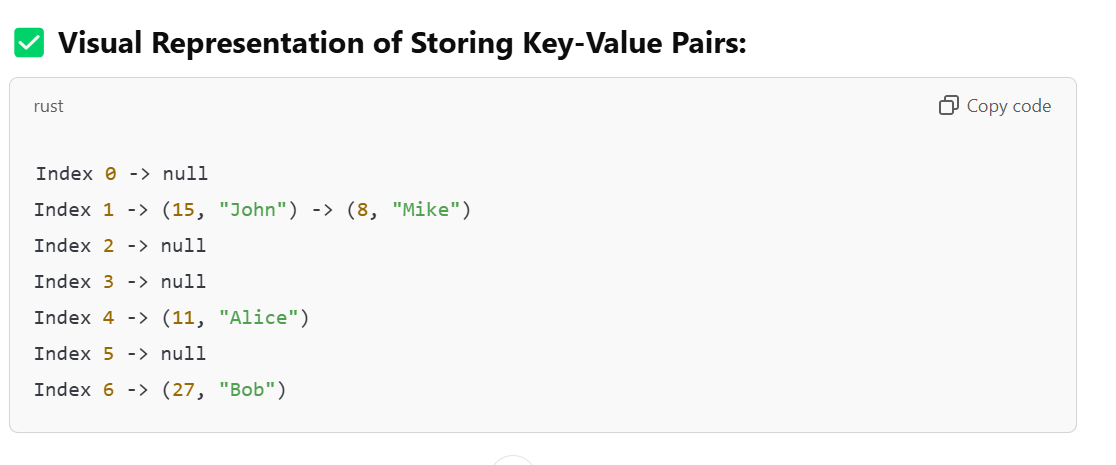
**Step 4: How HashMap Retrieves the Value**

When you try to **retrieve a value** using a key, the **HashMap** works like this:

1. **Apply the hash function** to the key to find the bucket (index).
2. **Traverse the LinkedList** in that bucket to find the key.
3. **Return the corresponding value**.

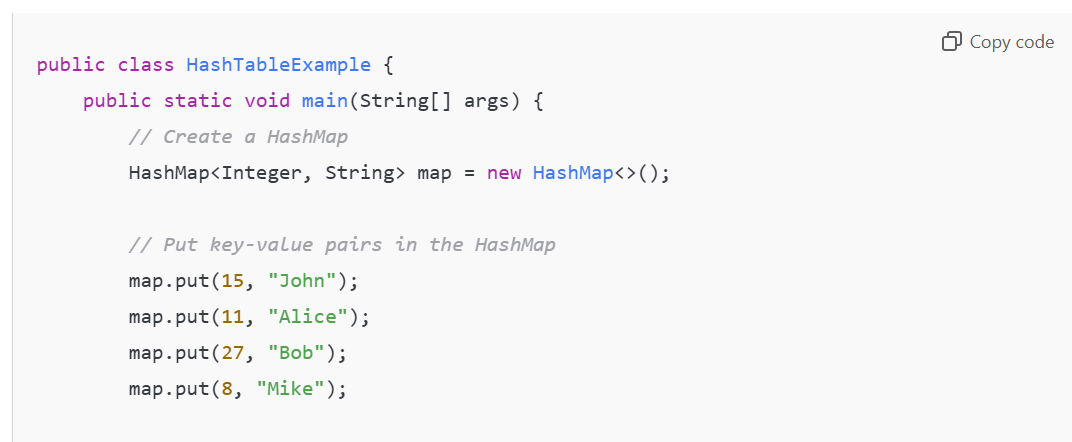
For example:

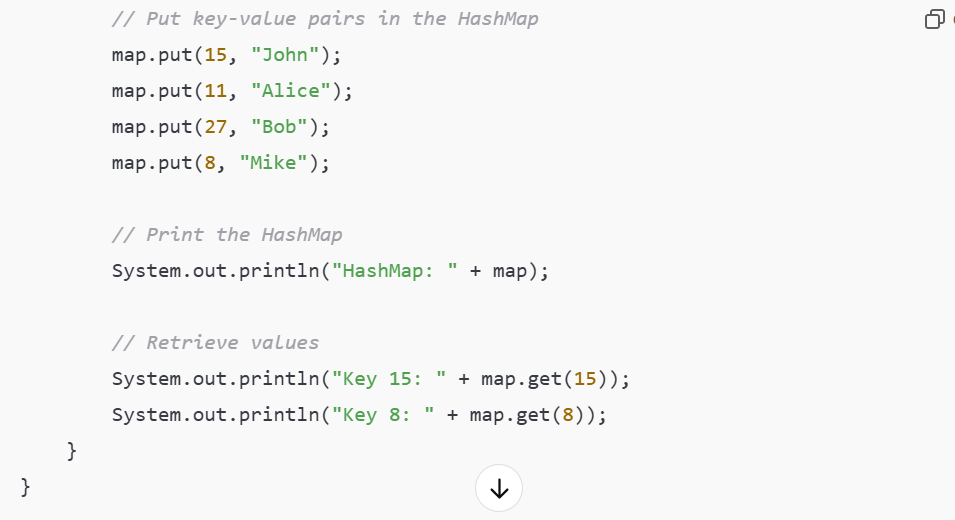
* If you search for **key 15**, the hash function gives **index 1**.
* At **index 1**, the LinkedList contains **(15, "John")** and **(8, "Mike")**.
* The HashMap compares the keys and finds **15**, then returns **"John"**.

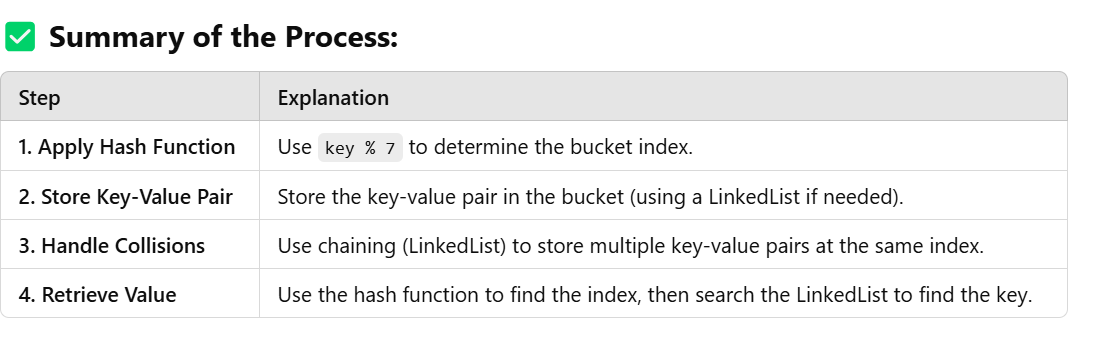


**Programmatic Example in Java:**

Here’s a **Java example** to demonstrate storing and retrieving key-value pairs in a HashMap.





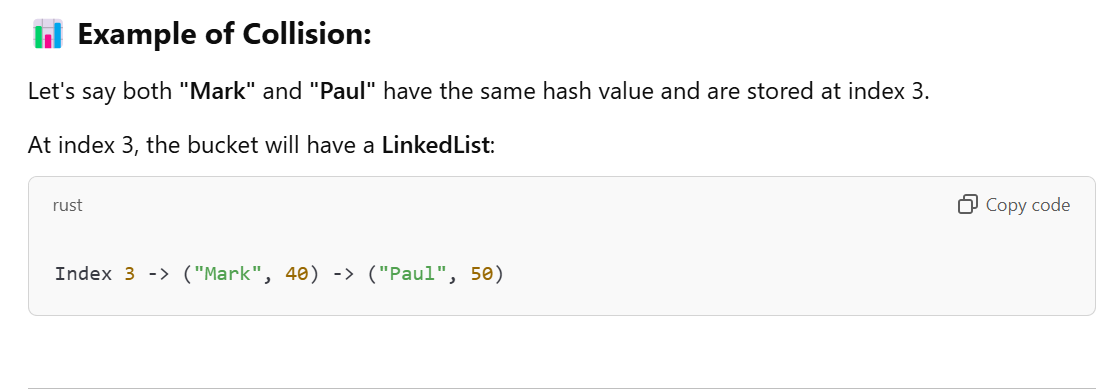
 

**Key Concepts to Remember:**

1. **Hash Function**:  
   A function that maps keys to a specific index in the hash table.  
   In this case: index = key % 7.
2. **Collision**:  
   When two keys are hashed to the same index.  
   Example: **15** and **8** both hashed to **bucket 1**.
3. **Collision Handling Using Chaining**:  
   A **linked list** is used at each bucket to handle multiple keys at the same index.

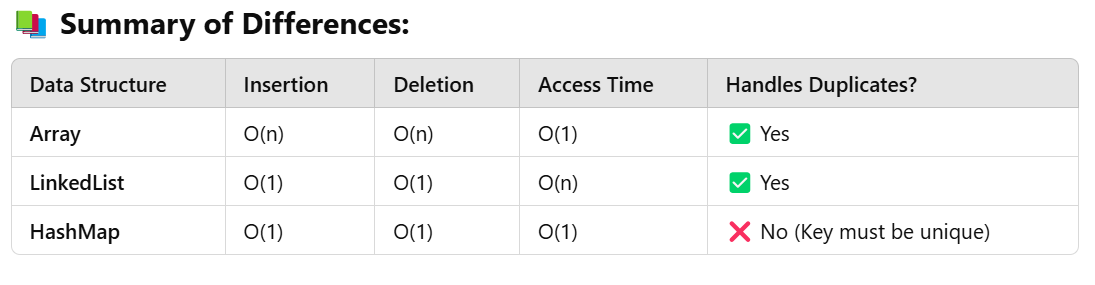
**What Happens if Two Keys Have the Same Index? (Collision)**

If two keys have the same index, it's called a **collision**.  
**HashMap** handles collisions using a **LinkedList** inside the bucket.



**Why Use Hashing?**

* **Fast Lookups**: Average time complexity is **O(1)**.
* **Efficient Storage**: Uses a combination of **array** and **LinkedList/tree**.



**What is a TreeSet in Java?**

A **TreeSet** is a part of **Java’s Collection framework** that implements the **Set interface** and is backed by a **TreeMap** (a type of self-balancing binary search tree).

**🔑 Key Properties of TreeSet:**

1. **No Duplicates Allowed** – Just like other Set implementations, **TreeSet** does not allow duplicate elements.
2. **Sorted Order** – **TreeSet stores elements in sorted (ascending) order**.
3. **No Null Elements Allowed** – TreeSet does not allow **null** values.
4. **Uses Red-Black Tree Internally** – TreeSet is implemented using a **self-balancing binary search tree** called a **Red-Black Tree**.

**⚙️ How TreeSet Works Internally?**

Internally, a **TreeSet** uses a **Red-Black Tree**, which is a **self-balancing binary search tree**. Here's how it works:

1. **Every element you add is stored in a node of the tree**.
2. The tree automatically maintains **sorted order** of elements.
3. The tree remains **balanced**, ensuring that all operations like **add**, **remove**, and **search** are efficient with a time complexity of **O(log n)**.

**🧩 How TreeSet Stores Data?**

Let’s say we add the following elements to a **TreeSet**:

makefile

Copy code

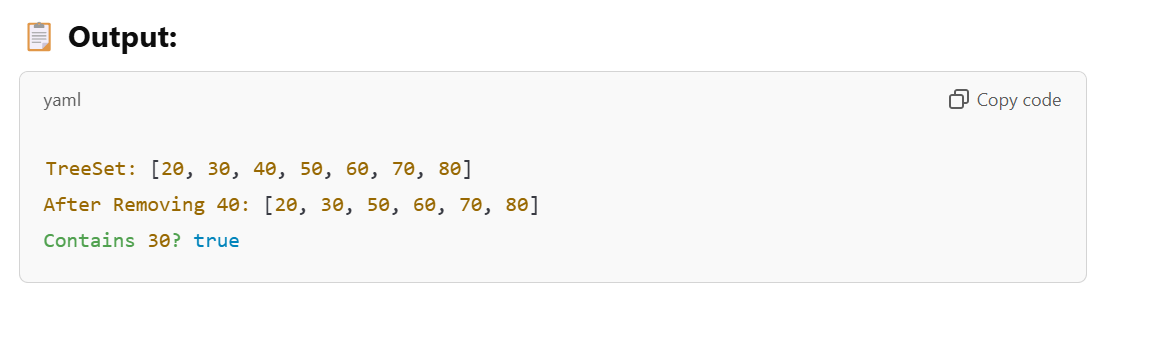
Elements: 50, 30, 70, 20, 40, 60, 80

The TreeSet will organize these elements in **sorted order** using a **binary search tree** (BST):





Java Example of TreeSet:



**What Happens When You Add Elements?**

Let's break down how elements are added to the TreeSet in **sorted order**.

**🔹 Adding 50:**

Since the tree is empty, **50** becomes the root node.

**🔹 Adding 30:**

**30** is smaller than **50**, so it goes to the **left** of **50**.

**🔹 Adding 70:**

**70** is greater than **50**, so it goes to the **right** of **50**.

**🔹 Adding 20:**

**20** is smaller than **50** and also smaller than **30**, so it goes to the **left of 30**.

The tree keeps adding elements in this manner while maintaining the **binary search tree property**.

**⚙️ How Does TreeSet Handle Duplicates?**

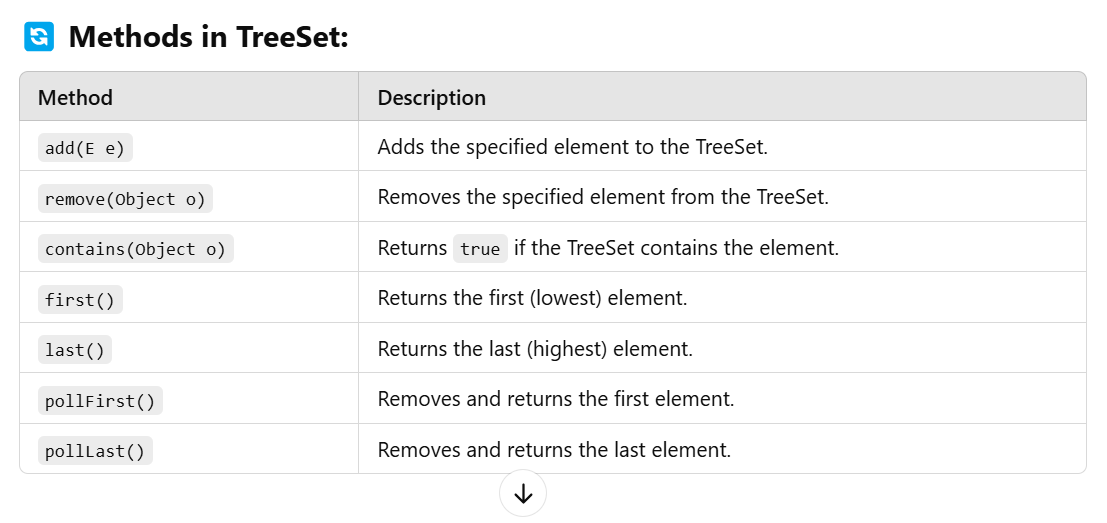
If you try to add a duplicate element, the TreeSet will simply **ignore** it.

**Example:**

java

Copy code







**Advantages of TreeSet:**

1. **Automatically sorts elements**.
2. Efficient for **range-based operations** (like finding elements within a specific range).
3. Provides methods to traverse in **both ascending and descending order**.

**🚫 Disadvantages of TreeSet:**

1. **Slower than HashSet** because of sorting.
2. **Does not allow null values**.
3. **More memory usage** compared to HashSet.

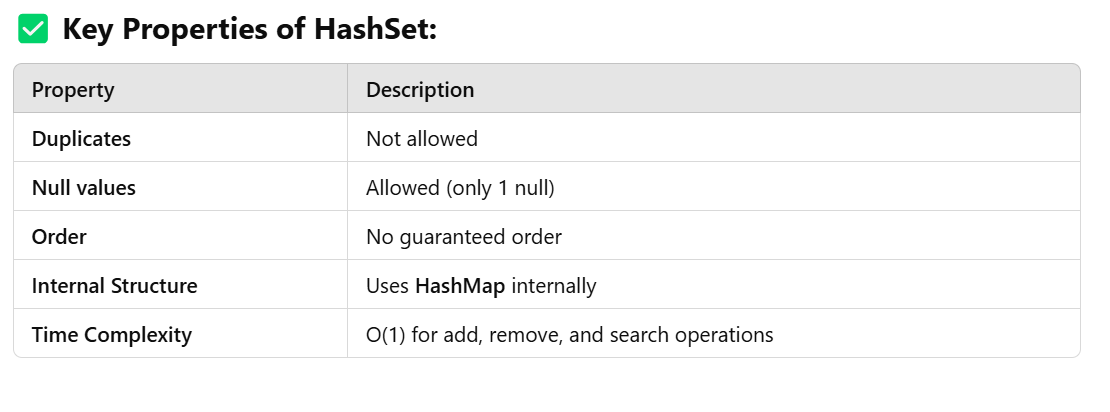
**Summary:**

1. **TreeSet** is a part of the **Set interface**.
2. It stores elements in **sorted order** using a **Red-Black Tree**.
3. It does **not allow null values** and ignores **duplicate elements**.
4. Operations like **add**, **remove**, and **search** take **O(log n)** time.
5. **Best for range-based queries** and when sorting is required.

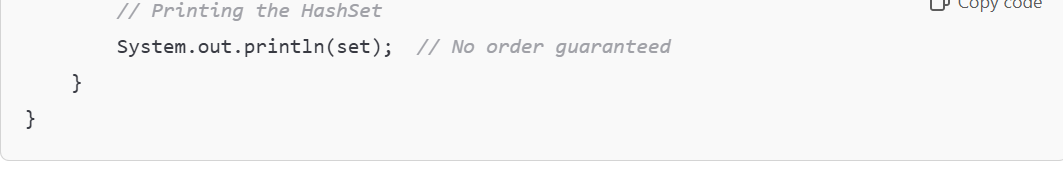
**HashSet in Java**

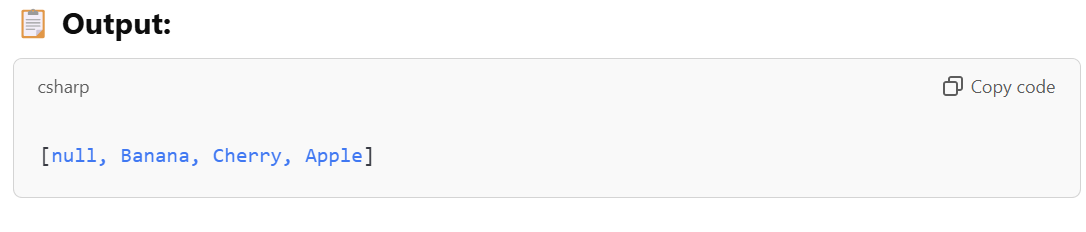
**HashSet** is a class that implements the **Set interface** and is backed by a **HashMap**. It **does not allow duplicate elements** and **does not maintain any order**.

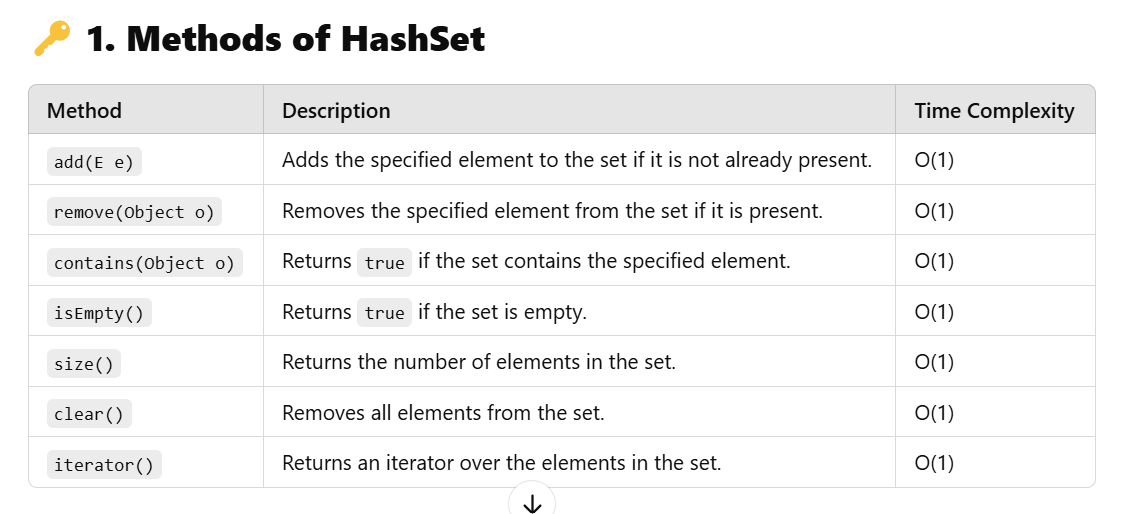
**✅ Key Properties of HashSet:**

****

****

****

****

****

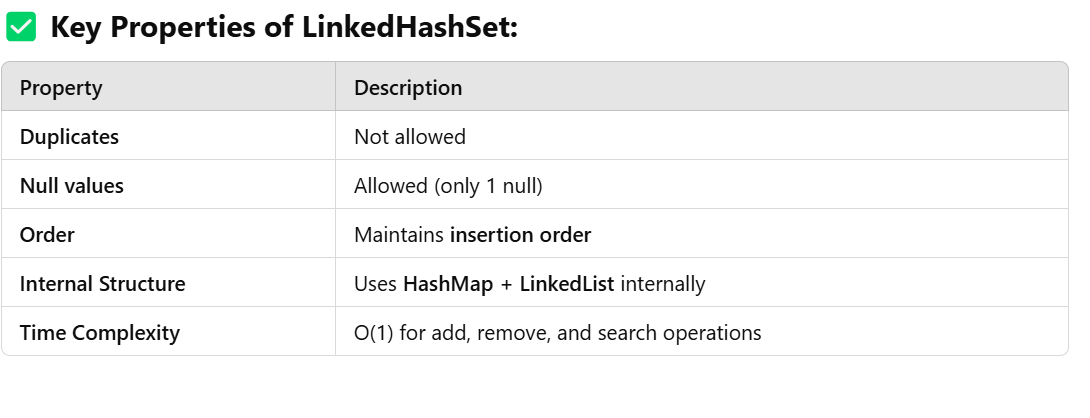
**When to Use HashSet?**

* **When you need to store unique elements and don’t care about order.**
* **For fast access operations.**

**LinkedHashSet in Java**

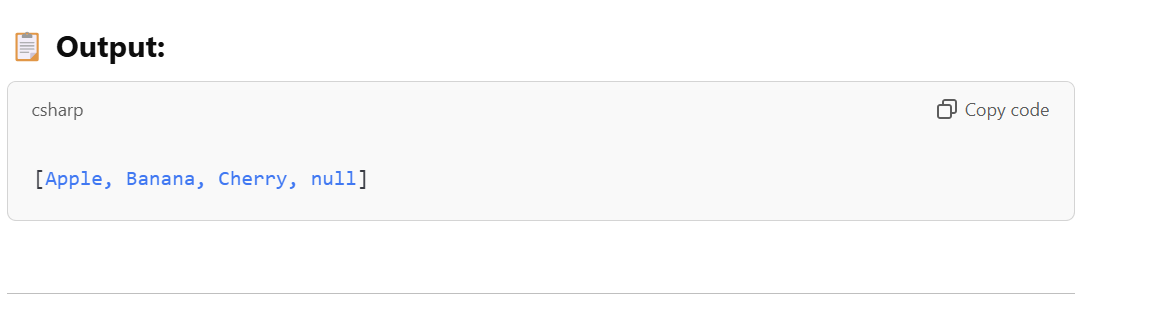
**LinkedHashSet** is a subclass of **HashSet** that **maintains the insertion order**. It uses a **combination of HashMap and LinkedList** internally.

**✅ Key Properties of LinkedHashSet:**

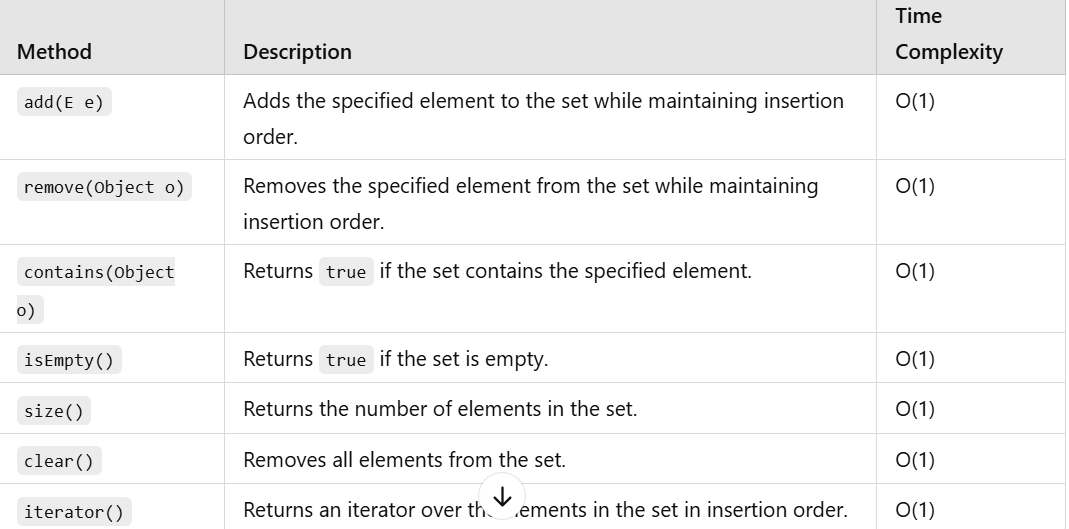


**Example of LinkedHashSet:**

****

****

1. **Methods of LinkedHashSet**

****

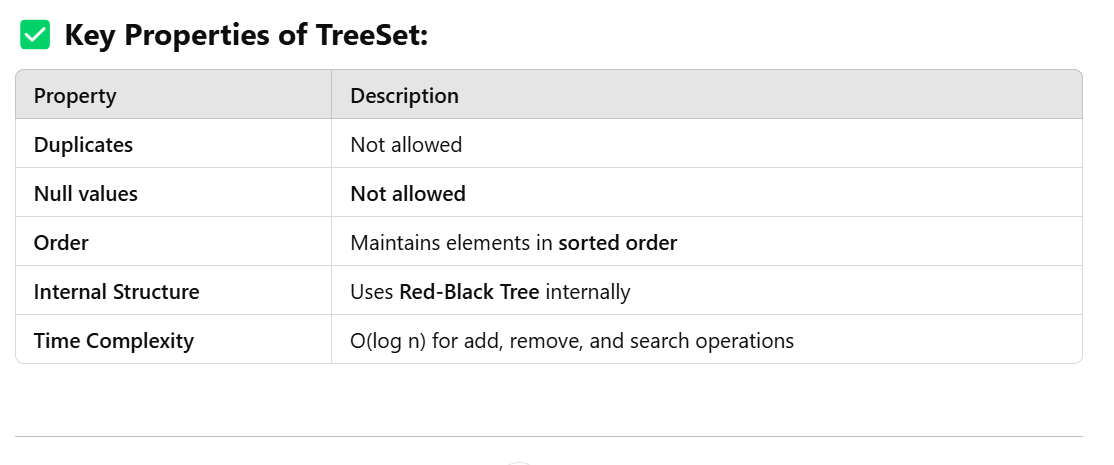
**When to Use LinkedHashSet?**

* When you need to store **unique elements** and **maintain the insertion order**.

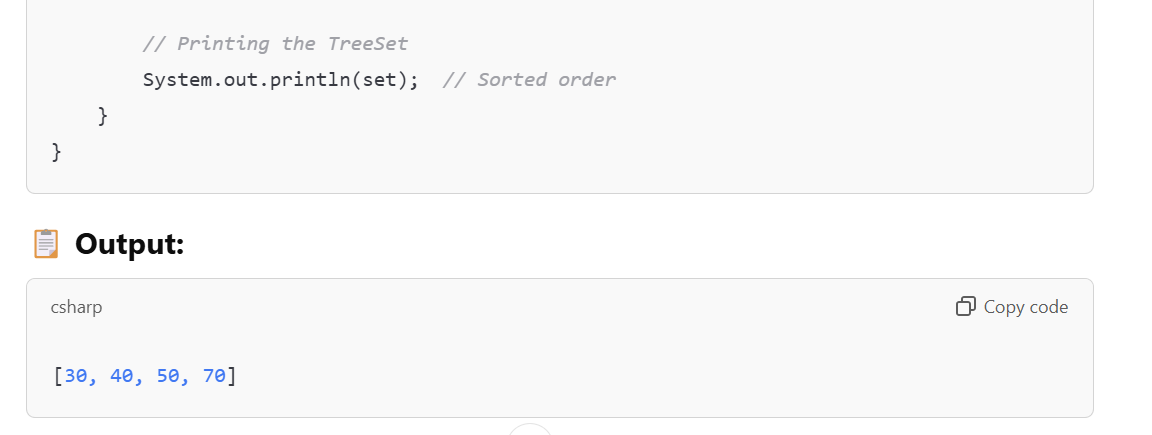
**🔎 TreeSet in Java**

**TreeSet is a class that implements the NavigableSet interface and is backed by a Red-Black Tree. It stores elements in sorted order.**

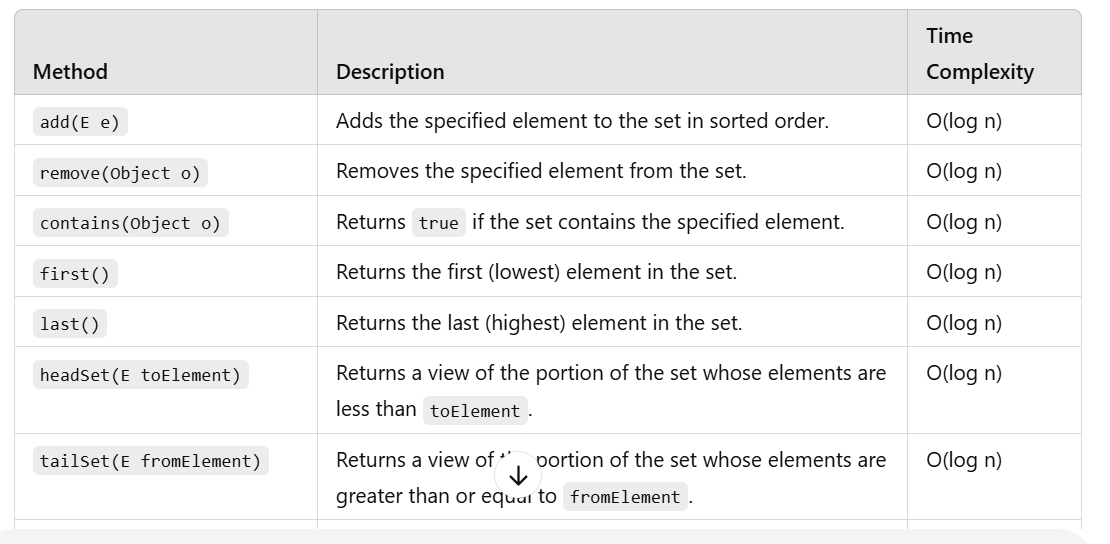
**✅ Key Properties of TreeSet:**

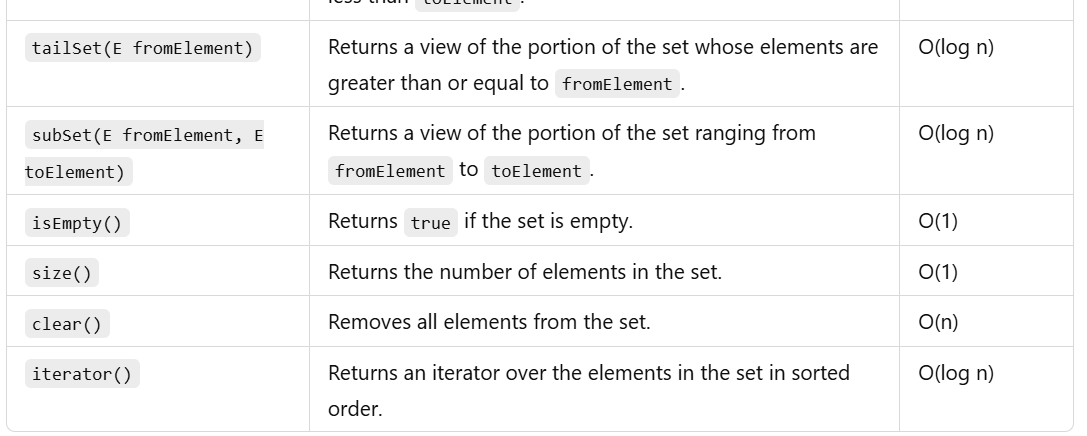
****

****

****

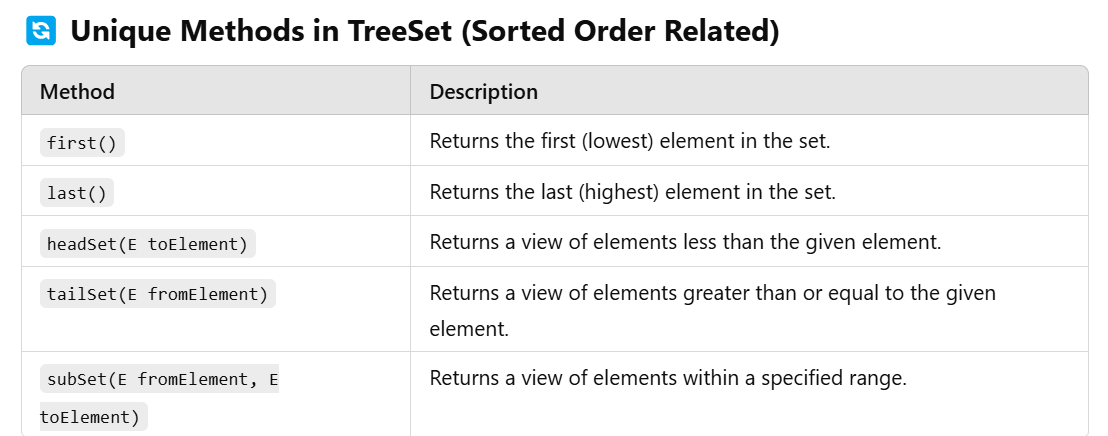
**Methods of TreeSet**

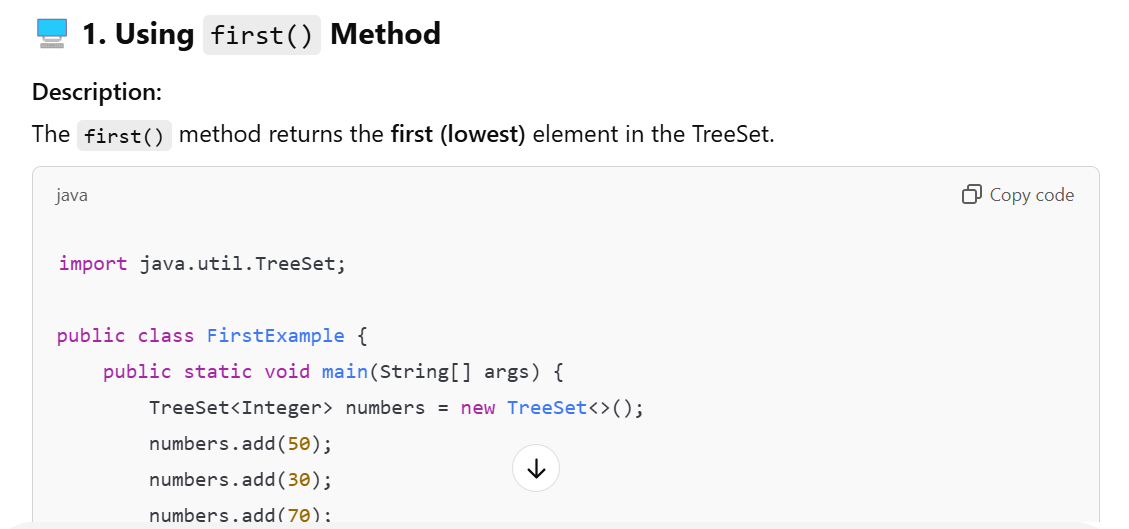




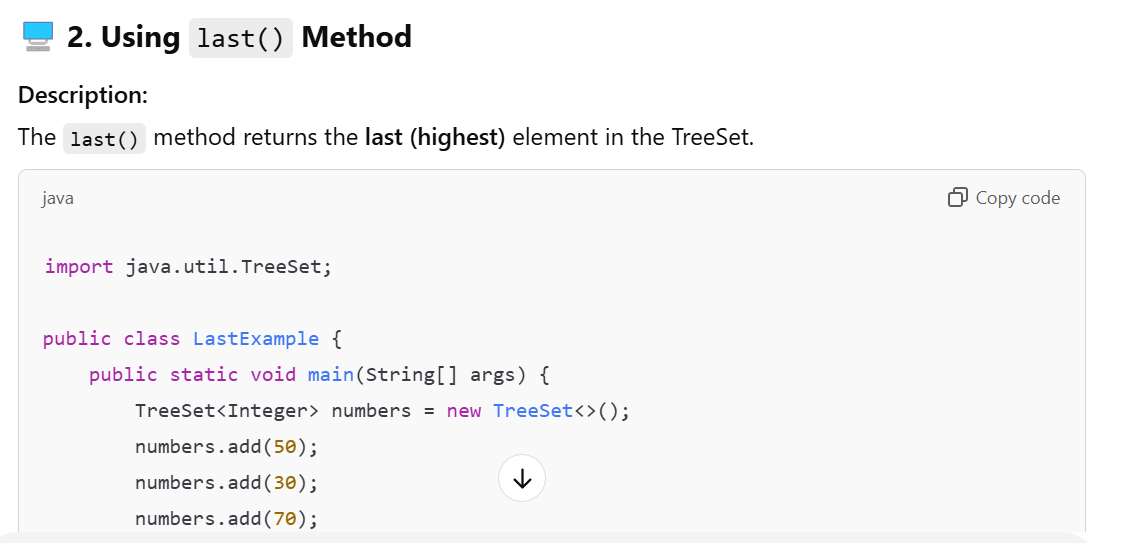
**When to Use TreeSet?**

* **When you need to store unique elements in sorted order.**

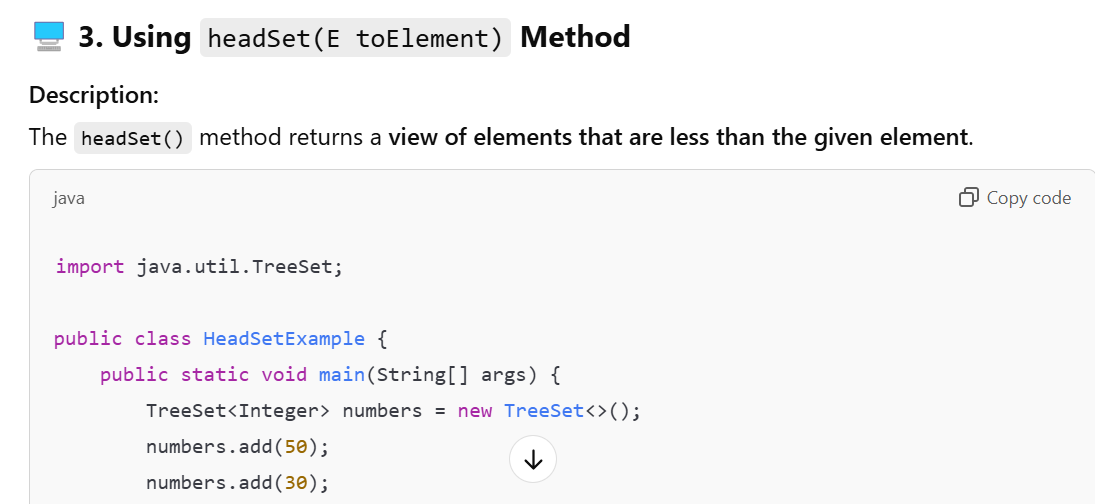
****

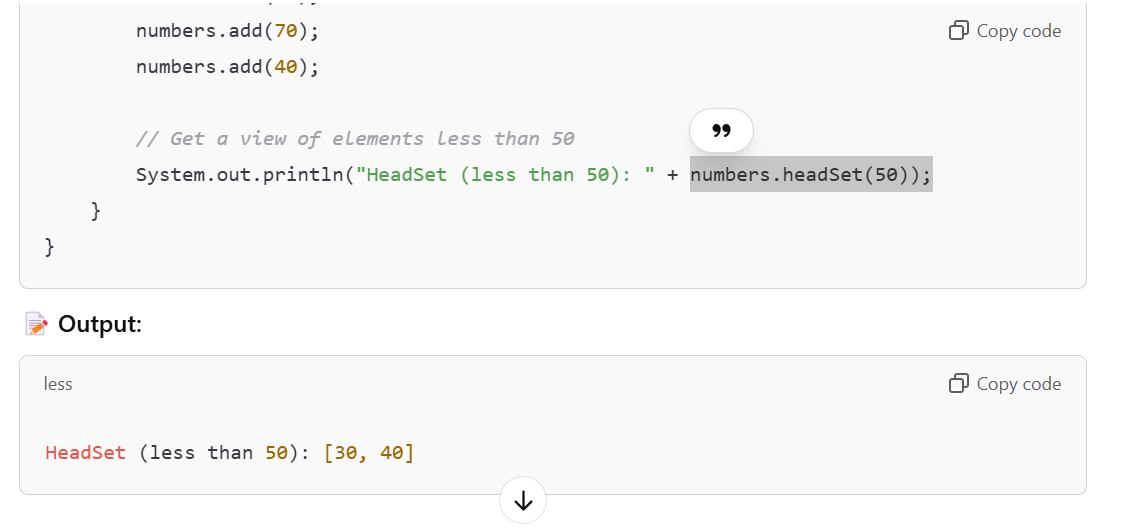
****

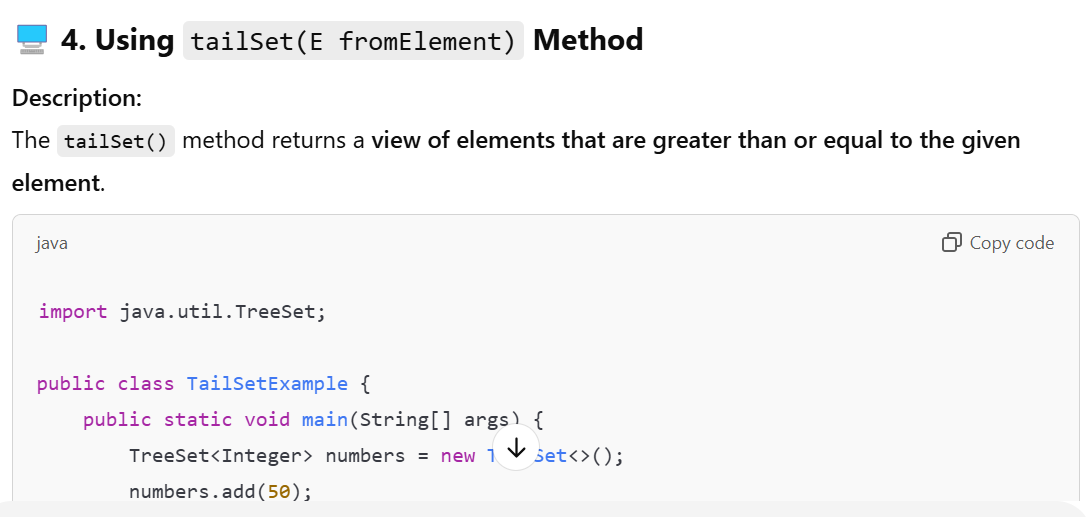
****

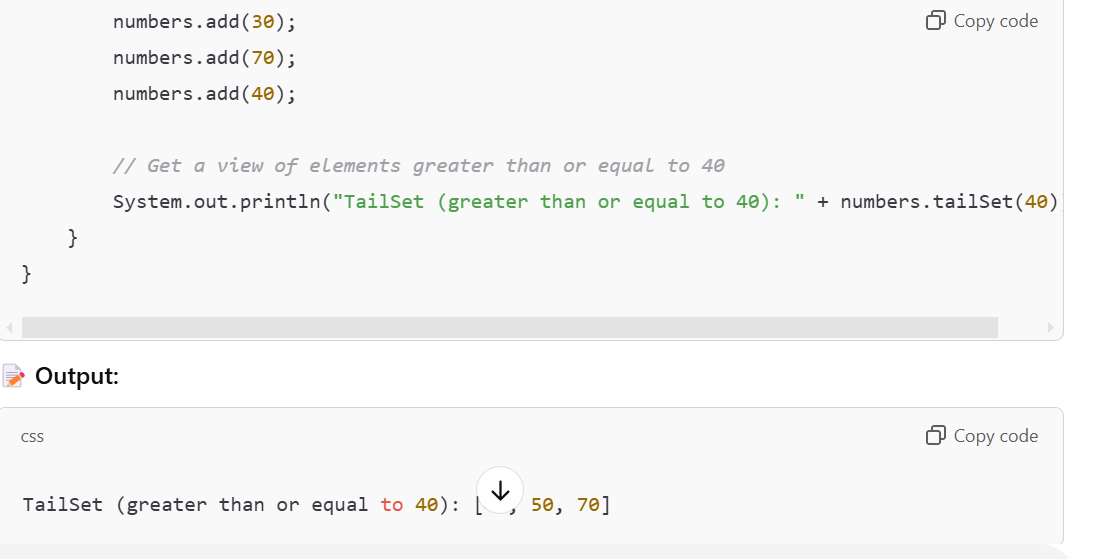
****

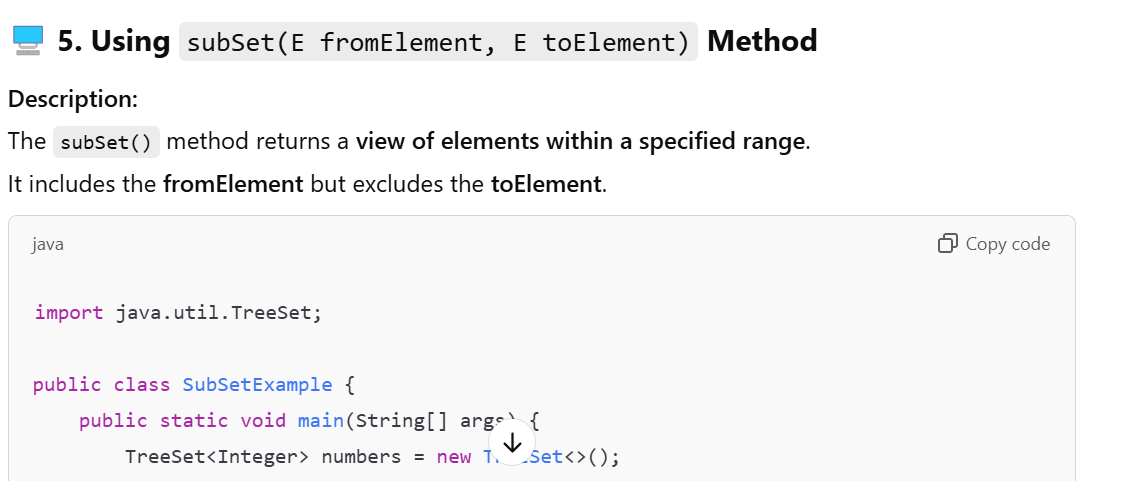
****

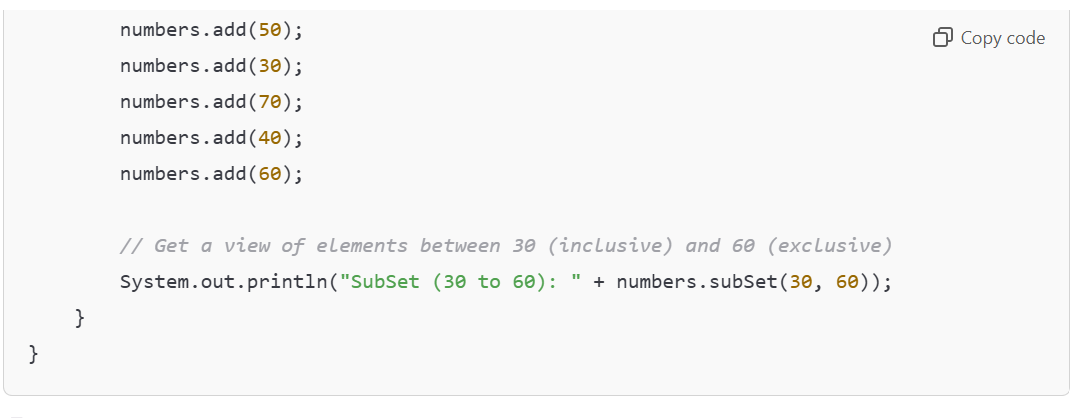
****

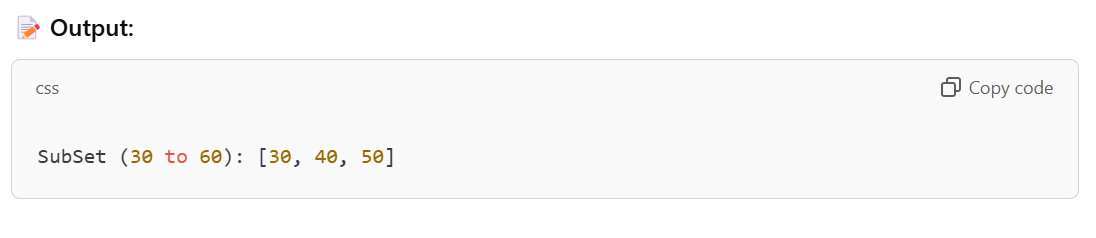
****

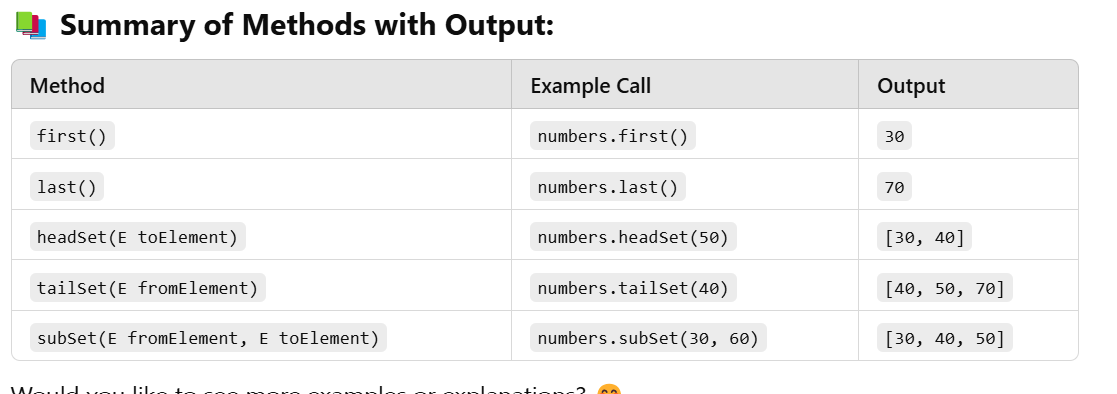
****

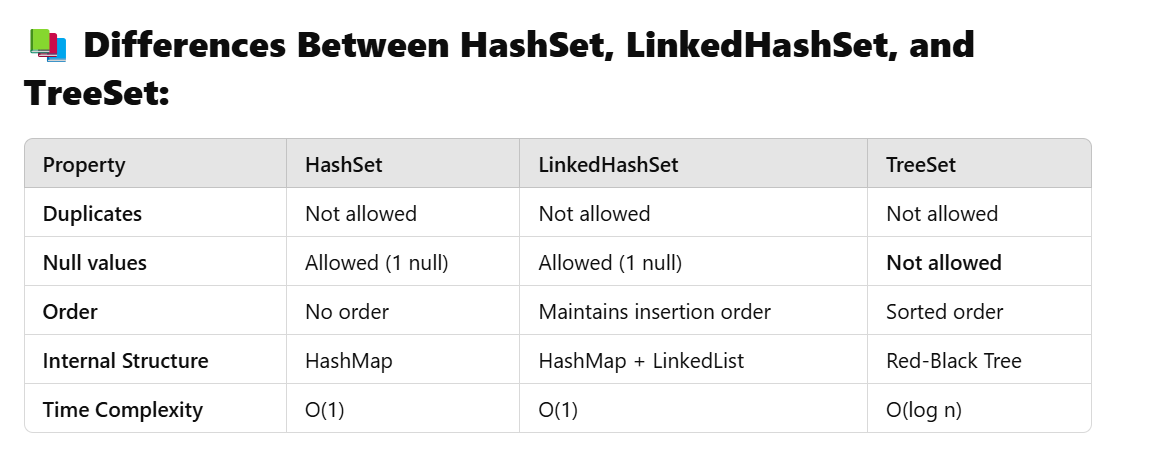
****

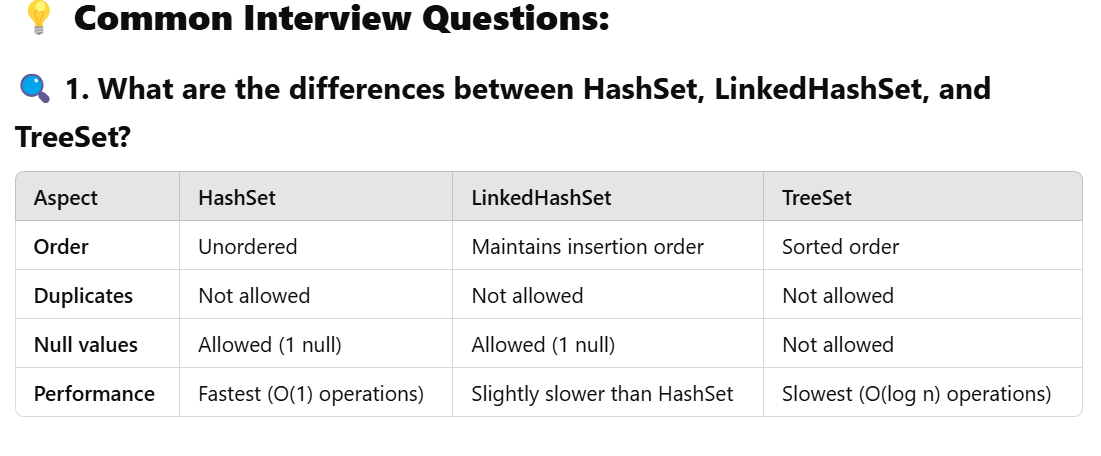
****

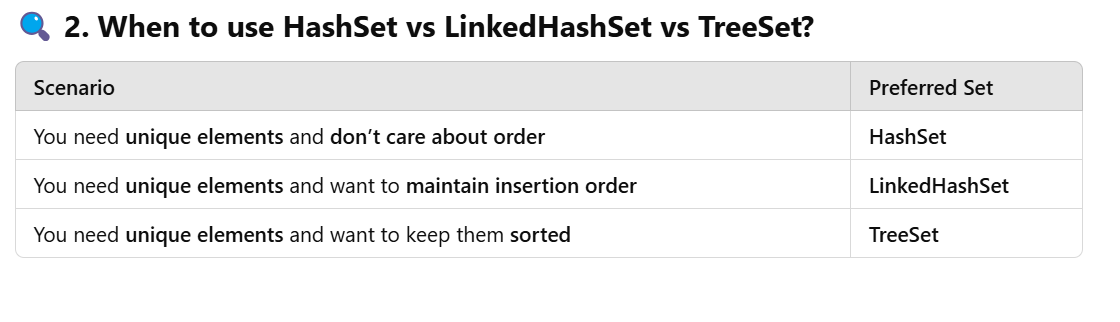
****

****

****

****

****

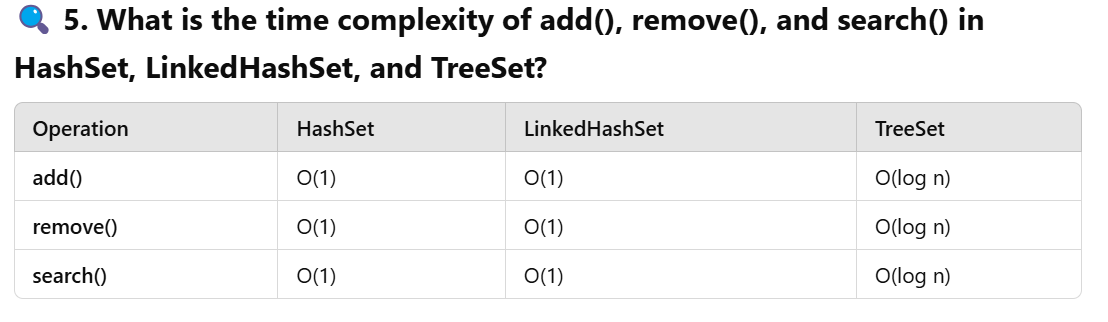
****

**3. Why doesn’t TreeSet allow null values?**

* TreeSet uses a **Red-Black Tree** internally, and it needs to compare elements to maintain **sorted order**.
* If you try to add a **null** value, the comparison will throw a **NullPointerException** because **null cannot be compared** to other elements.

**4. How does HashSet handle duplicates?**

* **HashSet** uses a **HashMap** internally.
* When you try to add a duplicate element, the **HashMap checks if the key already exists**.
* If the key exists, the new value is **ignored**.



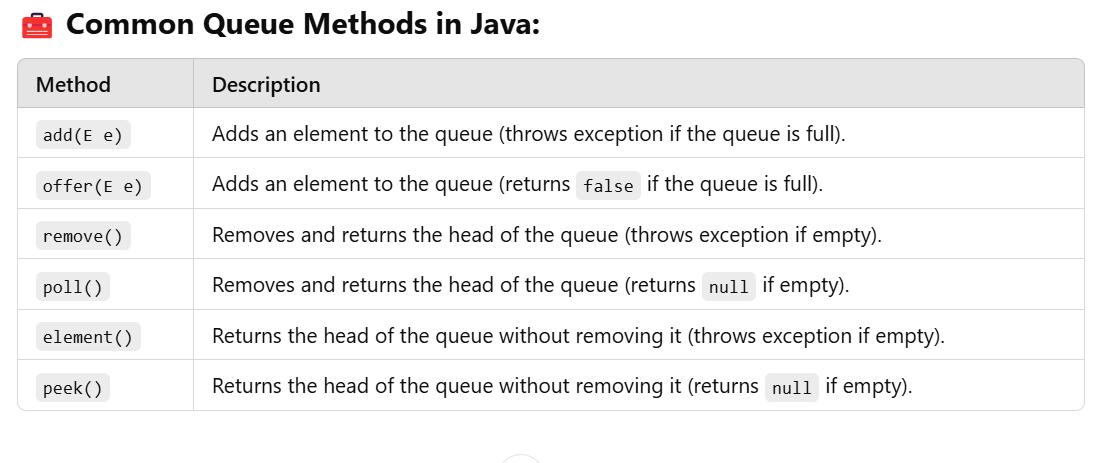
**1. What is a Queue in Java?**

**🔎 Definition**

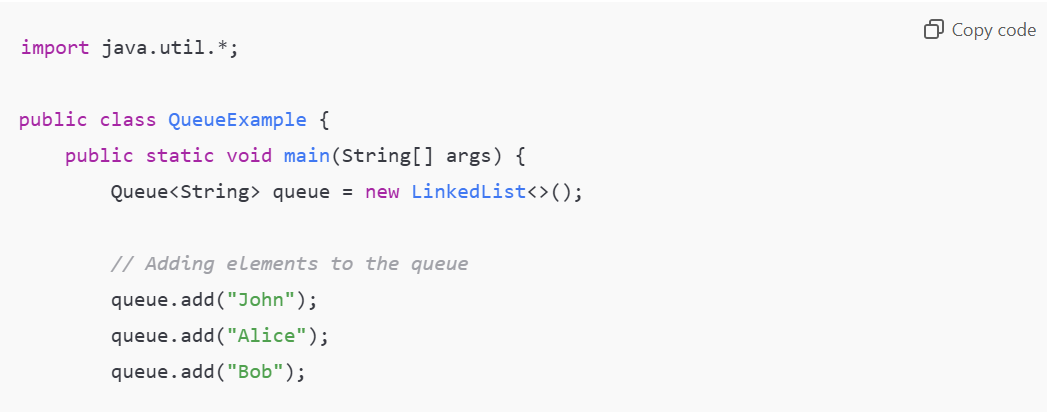
A **Queue** is a **linear data structure** that follows the **FIFO (First-In-First-Out)** principle. The first element added to the queue is the first one to be removed.

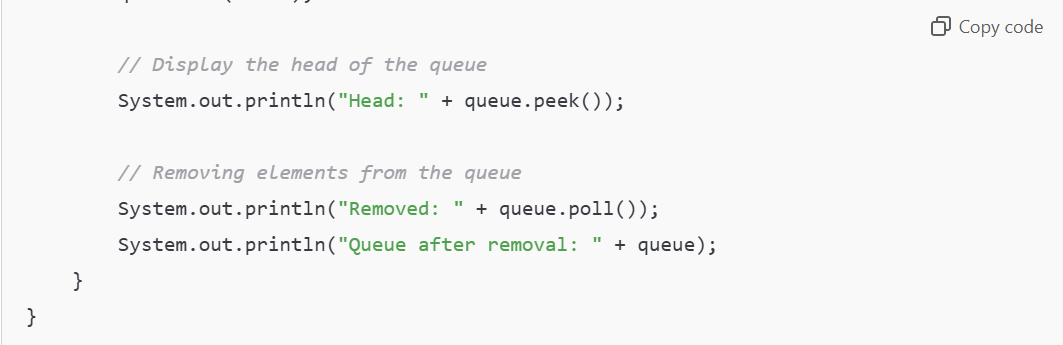
**⚙️ Key Characteristics of Queue:**

* **FIFO (First-In-First-Out)**
* Elements are added at the **rear** and removed from the **front**.
* **Null values are not allowed** in most queue implementations.



**Example of Queue Using LinkedList:**







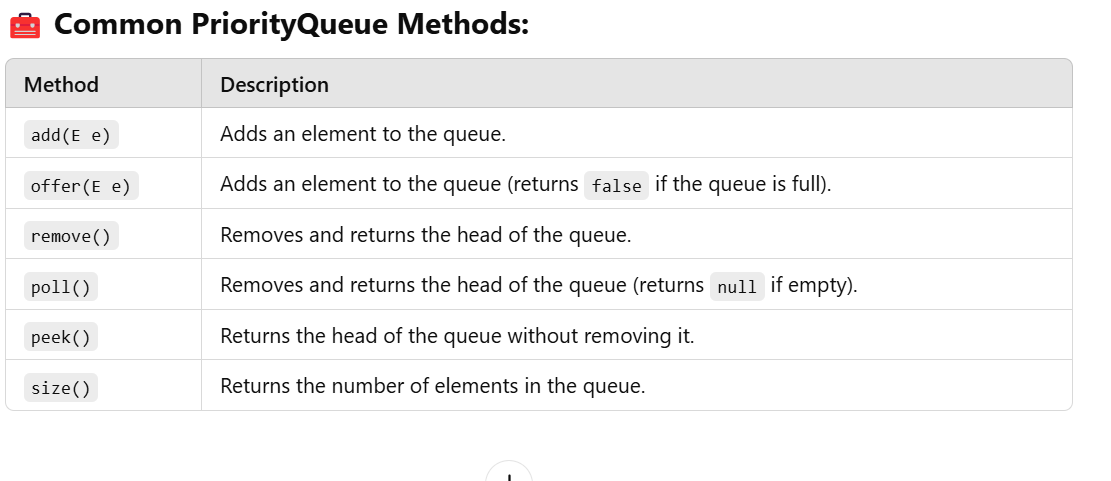
**2. What is a PriorityQueue in Java?**

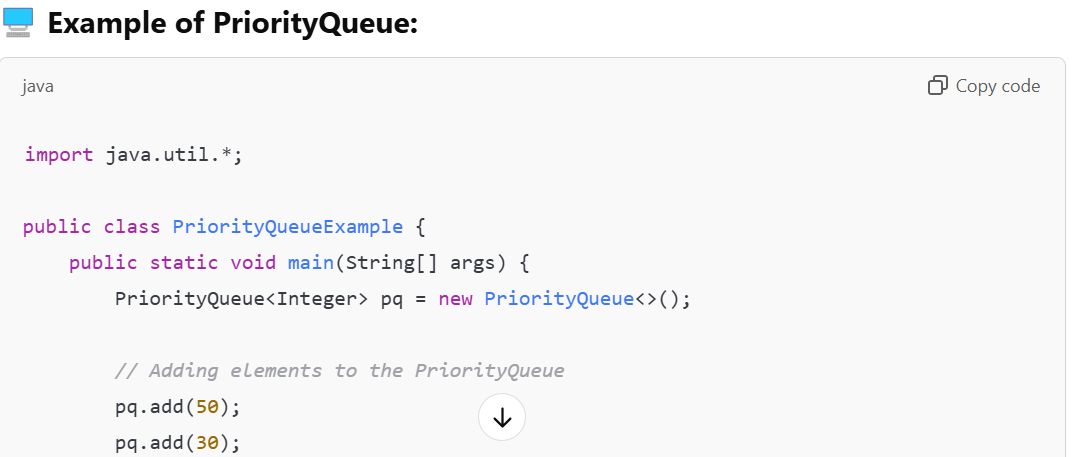
**Definition**

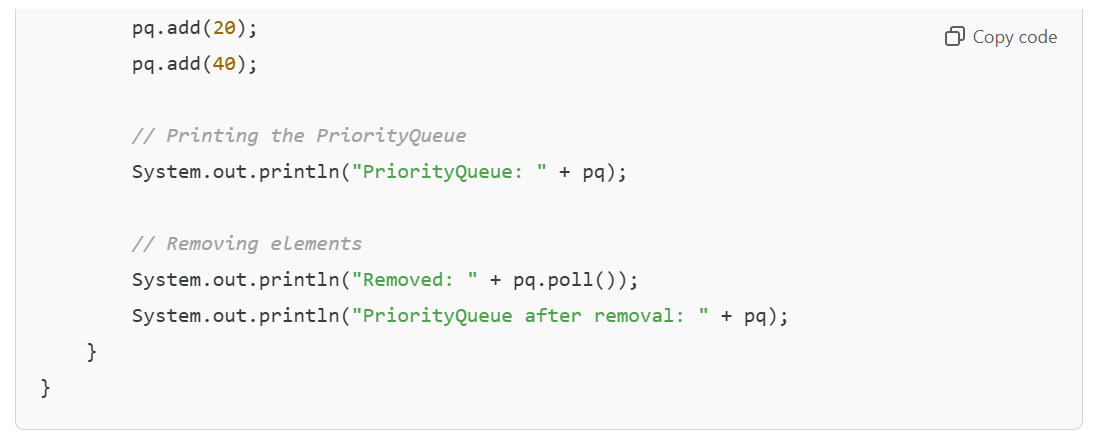
A **PriorityQueue** is a special type of queue in which **elements are ordered based on their natural ordering (ascending order)** or by a **custom comparator**.

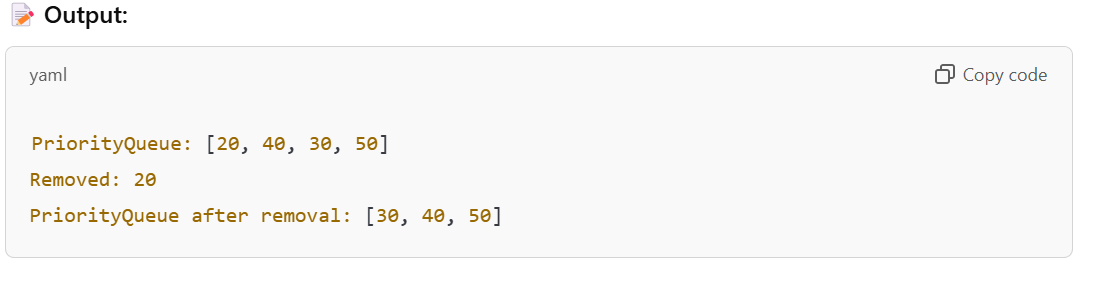
**⚙️ Key Characteristics of PriorityQueue:**

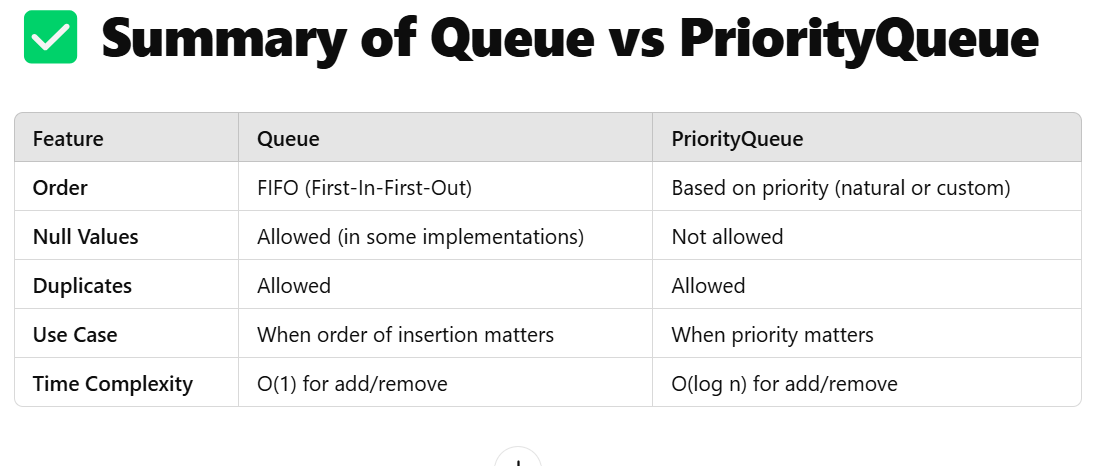
* Elements are ordered based on **priority**.
* It does **not follow FIFO**.
* **Null values are not allowed**.
* **Duplicates are allowed**.











**4.Custom PriorityQueue in Java**

**Definition**

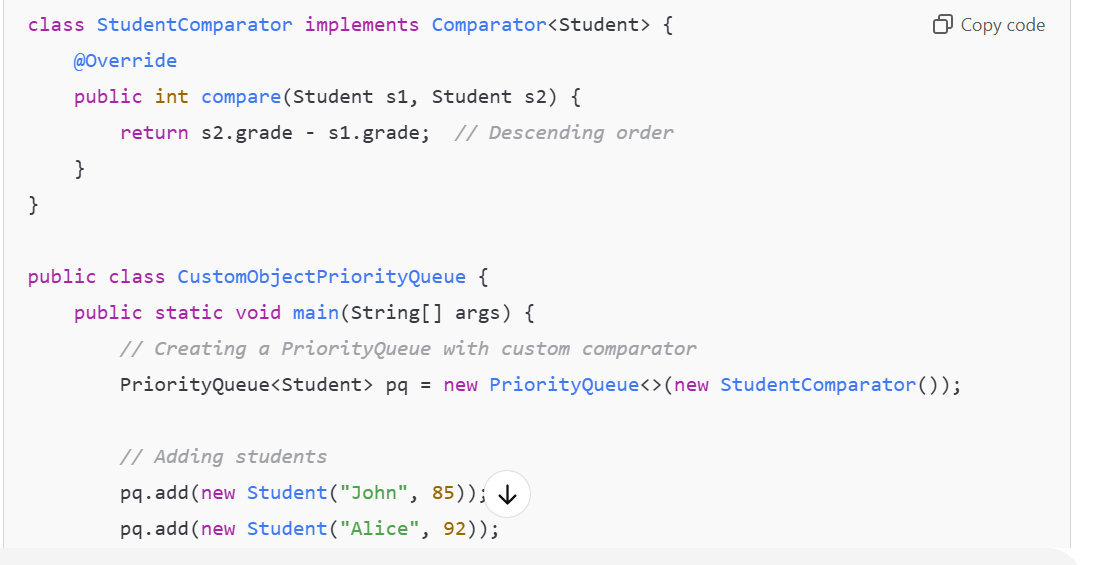
A **Custom PriorityQueue** allows you to define your **own priority rules** using a **Comparator**.

**🧰 How to Create a Custom PriorityQueue?**

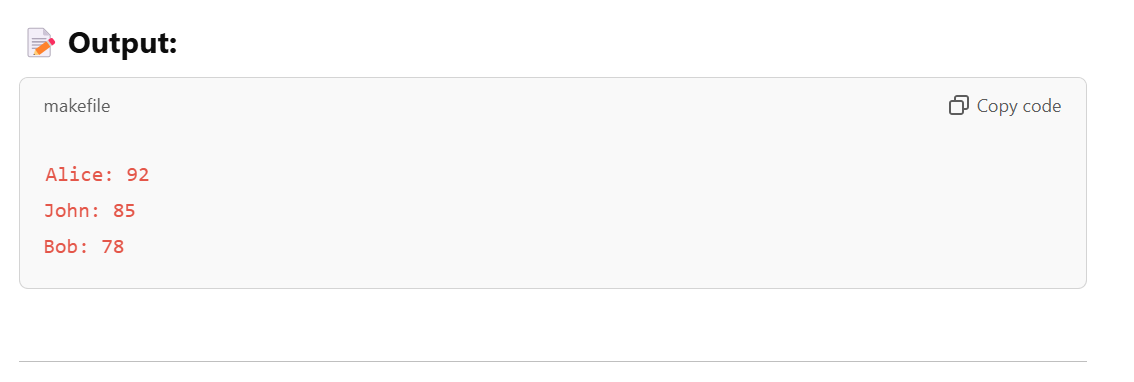
1. Use the **PriorityQueue** class and pass a **Comparator** to its constructor.
2. The **Comparator** defines the custom sorting logic.











**How the Code Works:**

1. **Student Class:**
   * Contains **name** and **grade** fields.
   * Overrides the **toString()** method for clean output.
2. **StudentComparator Class:**
   * Implements **Comparator<Student>**.
   * Compares **grades** in **descending order**.
3. **Main Class (CustomObjectPriorityQueue):**
   * Creates a **PriorityQueue** with the custom comparator.
   * Adds **Student objects** to the queue.
   * Uses **poll()** to retrieve and print elements in **priority order**.

**✅ Introduction to Map in Java**

The **Map interface** in Java is a part of the **Java Collections Framework**, but it is **not a subtype of the Collection interface**. Unlike other collections like **List** or **Set**, which store individual elements, **Map stores data in key-value pairs**.

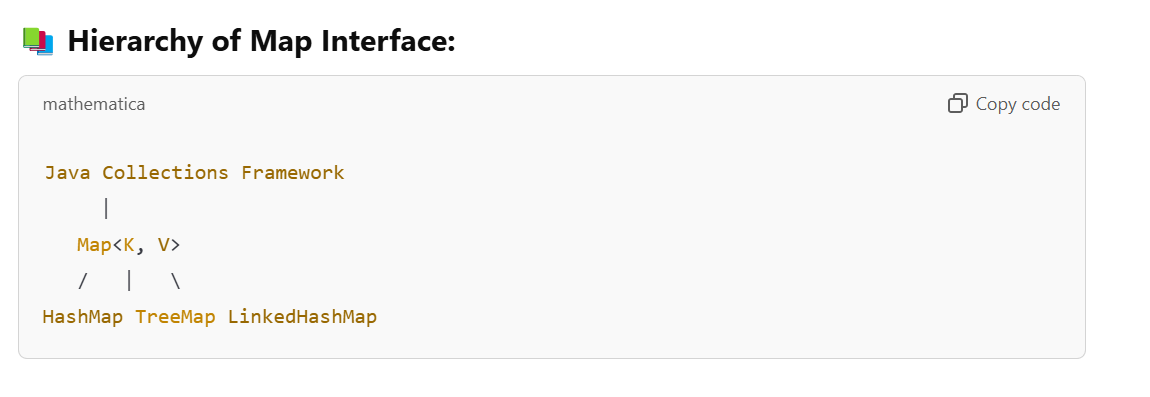
**🧩 Key Features of Map:**

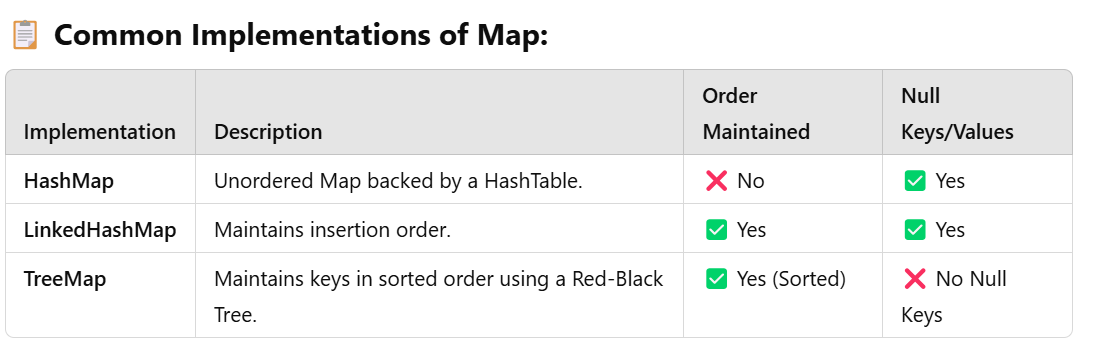
* **Map stores key-value pairs** where each key is unique.
* **Keys cannot be duplicate**, but **values can be duplicated**.
* A **key maps to a single value**, and each key-value pair is known as an **entry**.
* **Null keys** and **null values** are allowed, but behavior depends on the specific Map implementation.

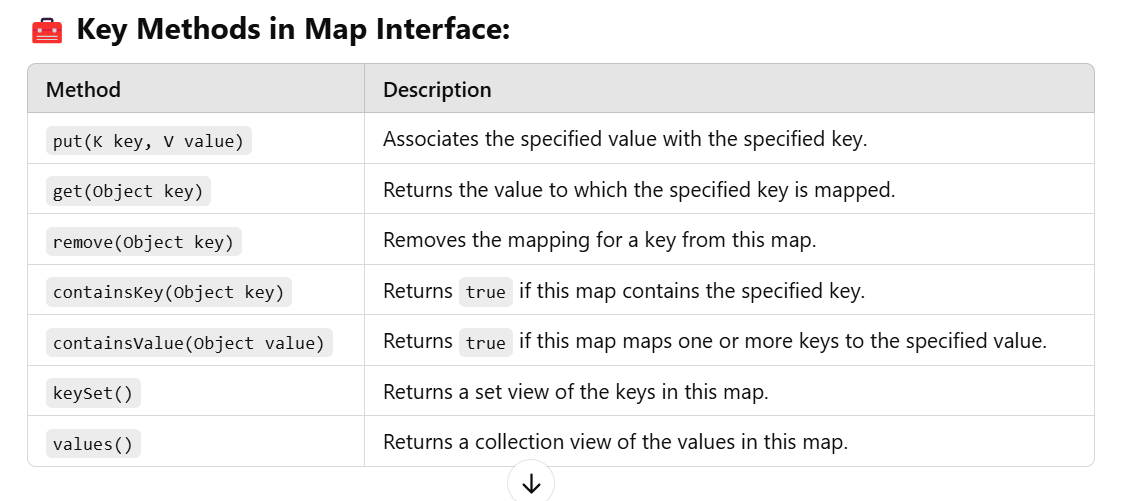
**⚙️ Why Map is Not a Collection?**

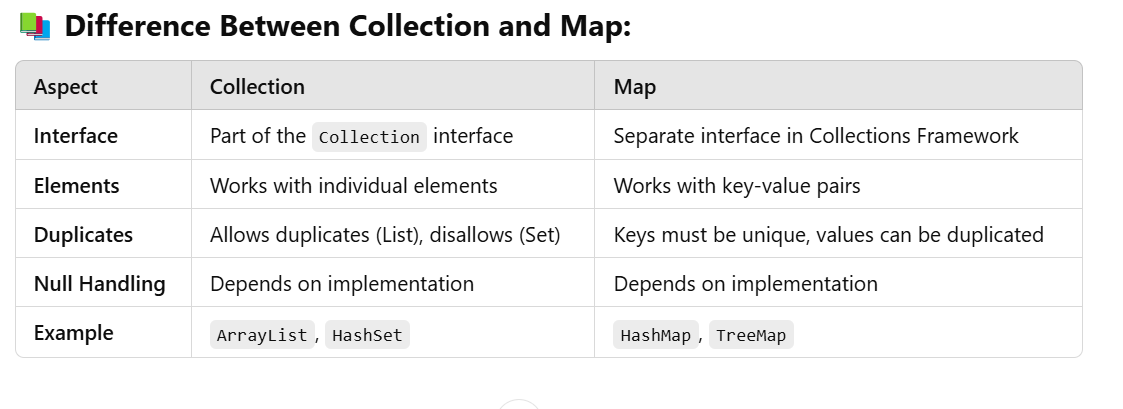
Although **Map** is part of the **Collections Framework**, it is **not a subtype of the Collection interface** because:

1. **Collection works with individual elements**, while **Map works with key-value pairs**.
2. **Collection methods like add(), remove(), etc., don’t make sense for a Map**.









Here’s a complete explanation of **HashMap**, **TreeMap**, **LinkedHashMap**, and **Hashtable** in Java with their **methods**, **examples**, and **common interview questions** to help you prepare for interviews.

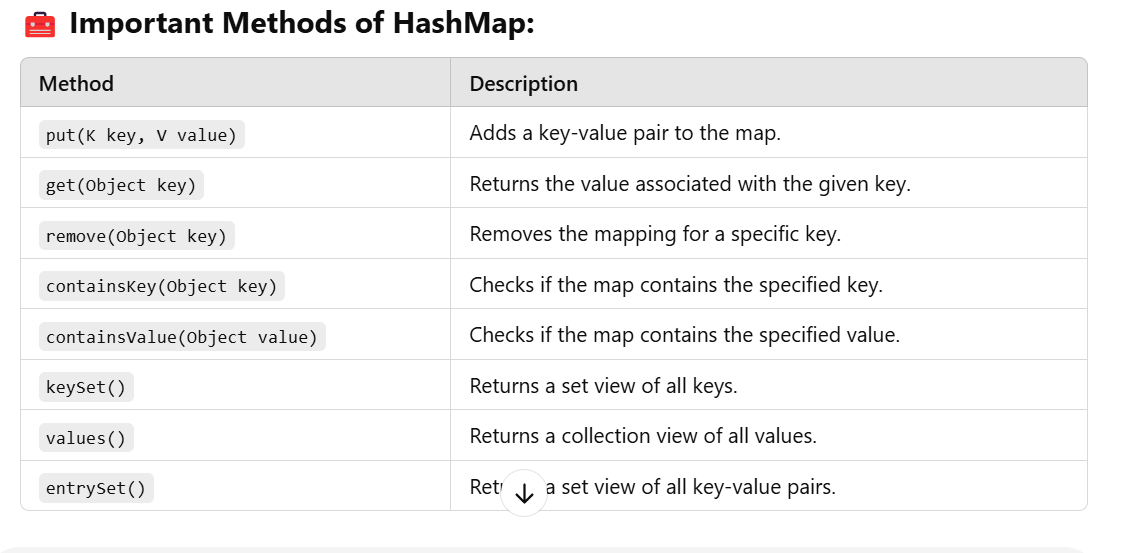
**📚 1. HashMap in Java**

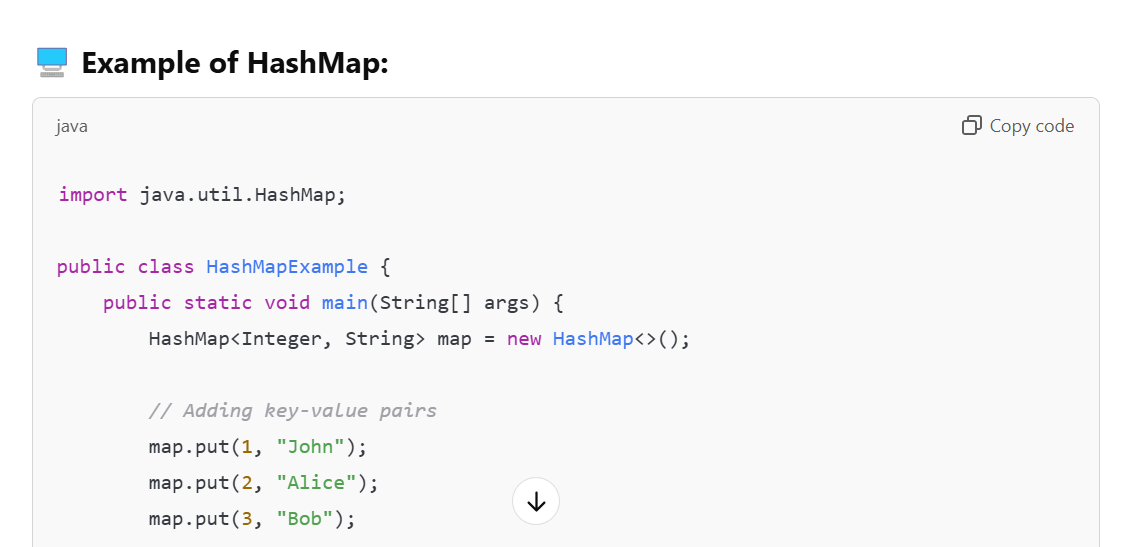
**🔎 What is HashMap?**

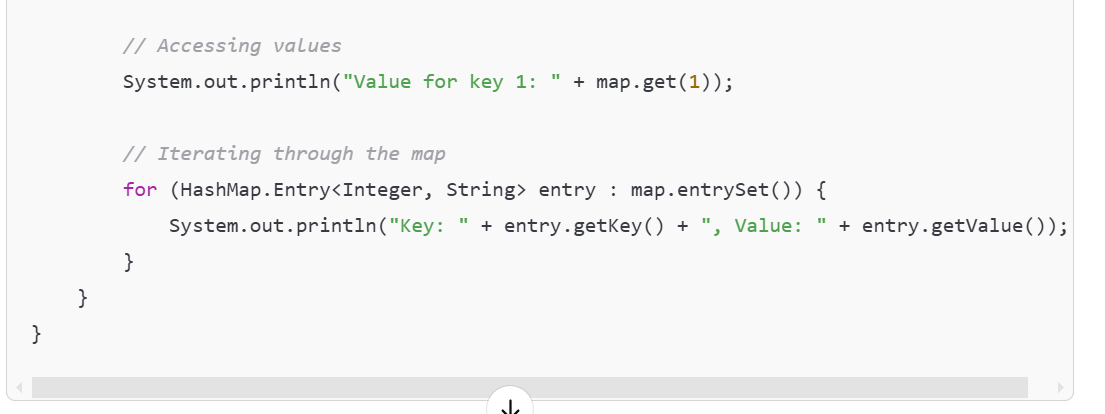
**HashMap** is a class that implements the **Map interface** and stores **key-value pairs**. It uses a **HashTable** internally to store data and provides **constant-time performance (O(1))** for basic operations like **add**, **remove**, and **get**.

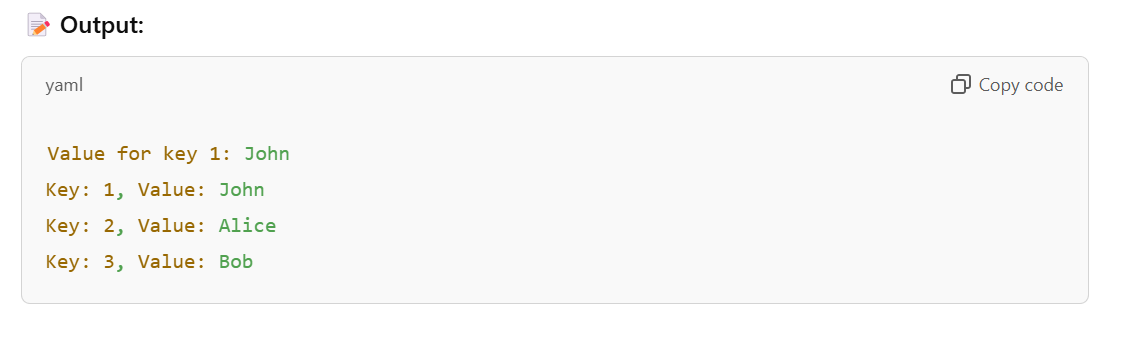
**⚙️ Key Features of HashMap:**

* **No order is maintained**.
* **Allows one null key** and **multiple null values**.
* **Fast performance** for large datasets.
* **Not thread-safe**.









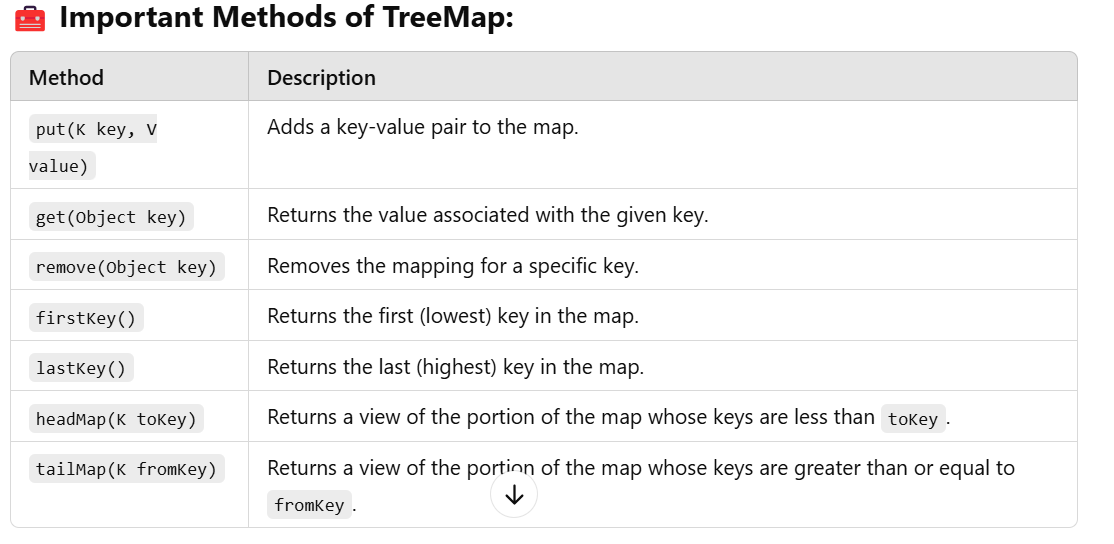
**📚 2. TreeMap in Java**

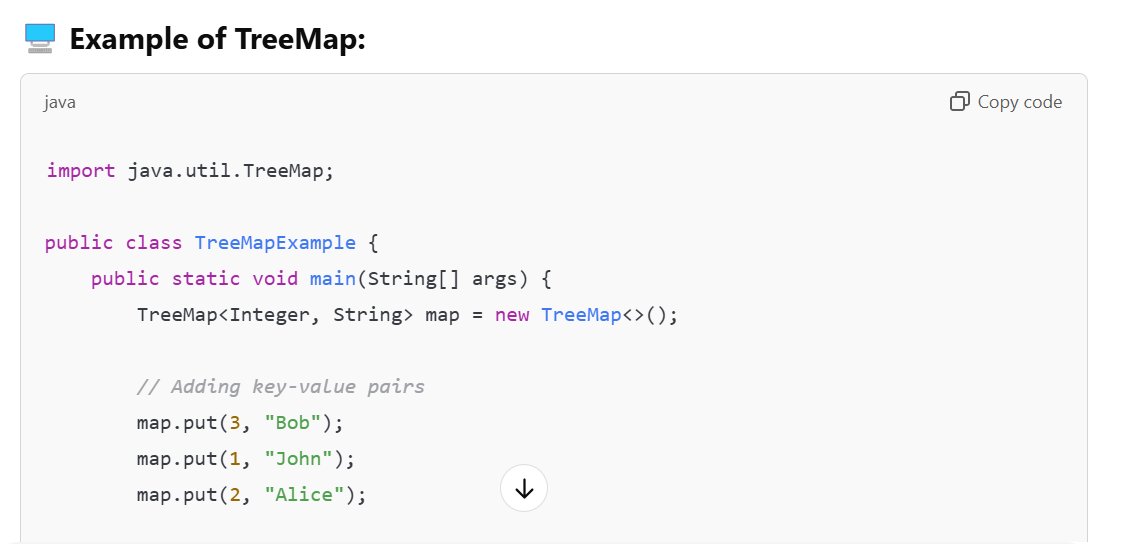
**🔎 What is TreeMap?**

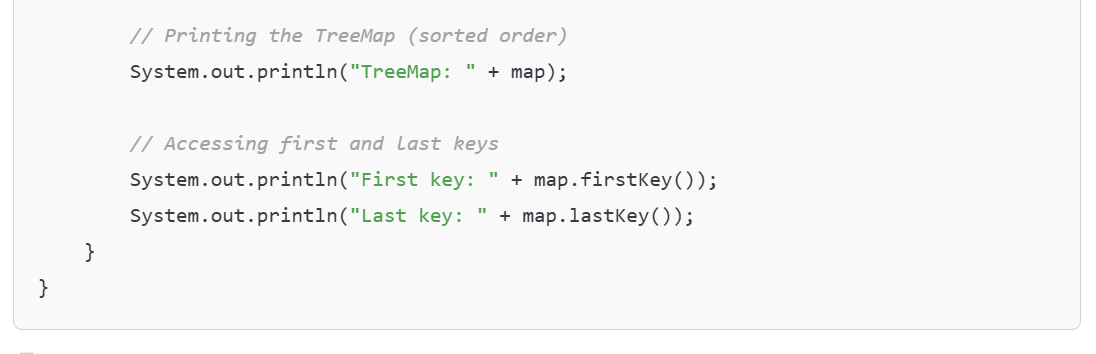
**TreeMap** is a class that implements the **Map interface** and stores **key-value pairs** in **sorted order** based on the keys. It uses a **Red-Black Tree** internally to maintain the order.

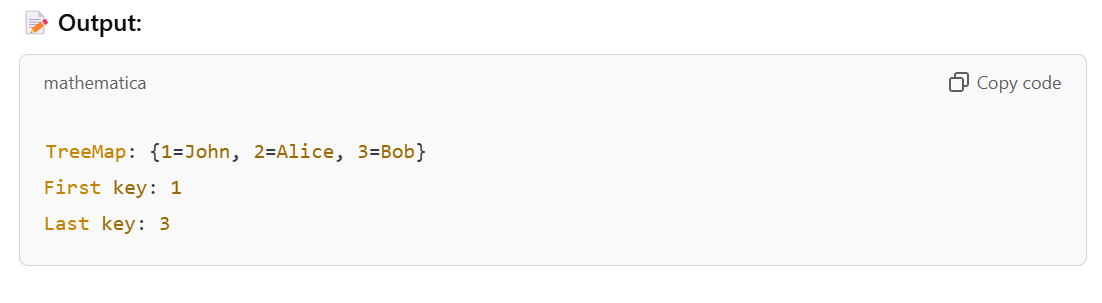
**⚙️ Key Features of TreeMap:**

* **Maintains natural order** (or custom order using a comparator).
* **Does not allow null keys**, but **allows null values**.
* **Thread-safe version available through Collections.synchronizedMap()**.









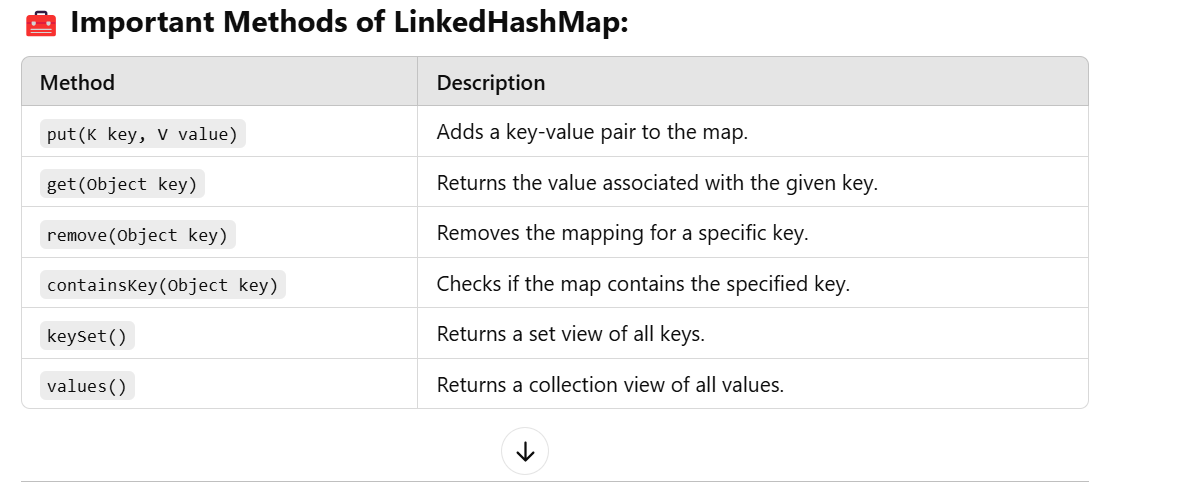
📚 **3. LinkedHashMap in Java**

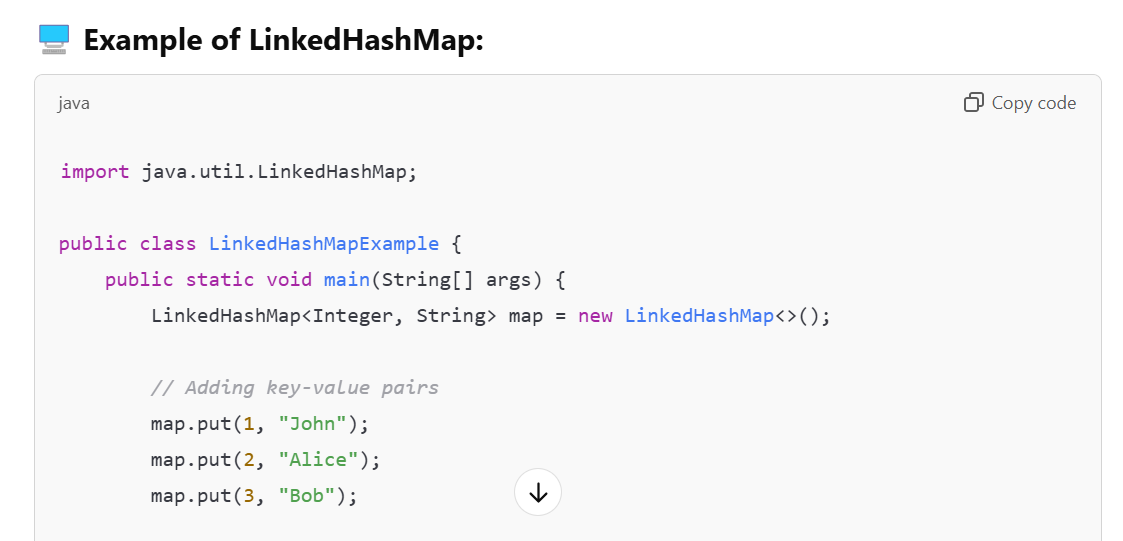
**🔎 What is LinkedHashMap?**

**LinkedHashMap** is a class that implements the **Map interface** and stores **key-value pairs** while maintaining the **insertion order**.

**⚙️ Key Features of LinkedHashMap:**

* **Maintains insertion order**.
* **Allows one null key** and **multiple null values**.
* **Faster than TreeMap**, but **slower than HashMap**.



**📚 4. Hashtable in Java**

**🔎 What is Hashtable?**

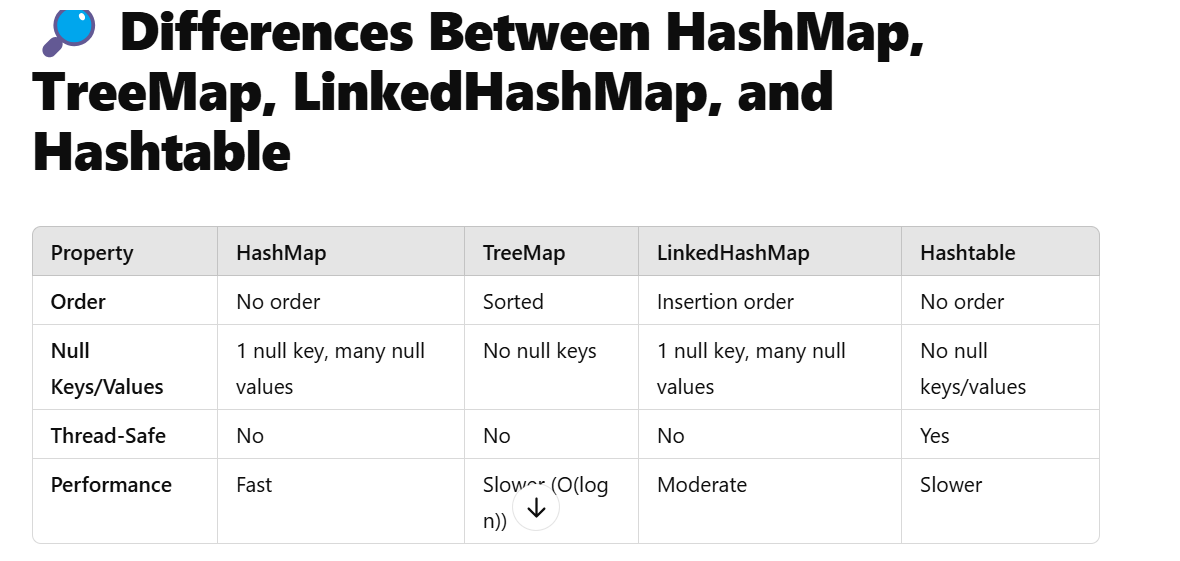
**Hashtable** is a class that implements the **Map interface** and stores **key-value pairs** in a **synchronized** manner. It is similar to **HashMap**, but it is **thread-safe**.

**⚙️ Key Features of Hashtable:**

* **Thread-safe** (all methods are synchronized).
* **Does not allow null keys or null values**.
* **Slower than HashMap**.





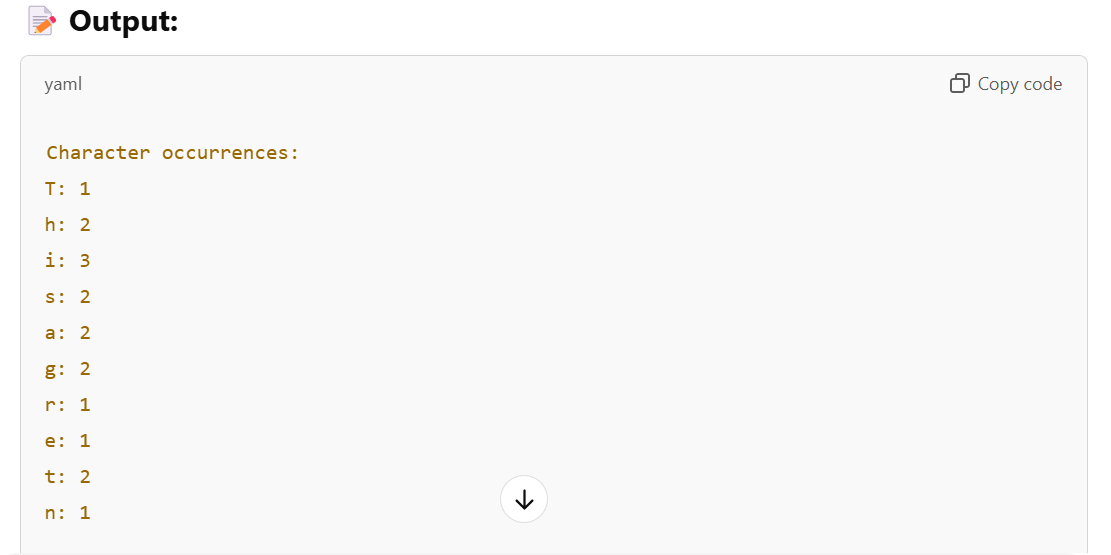
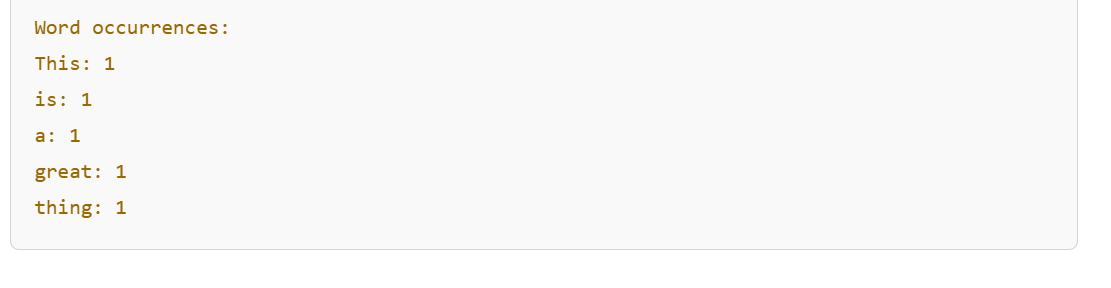


**Question : String = "This is a great thing" count the occurace of the character and word in**



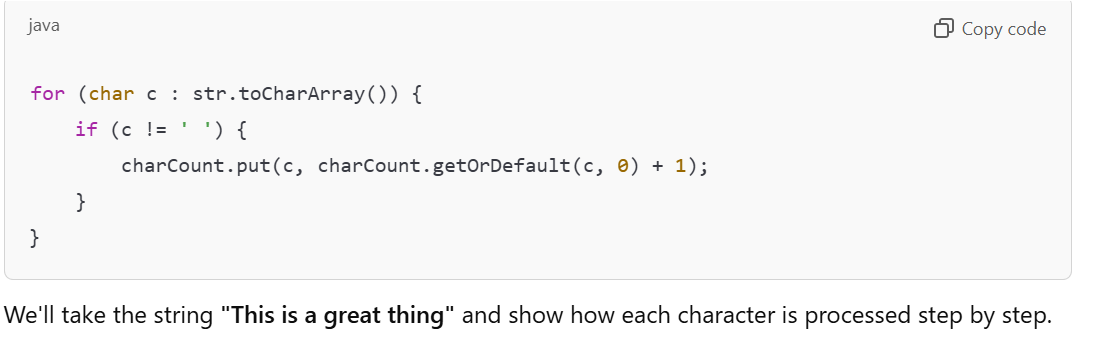




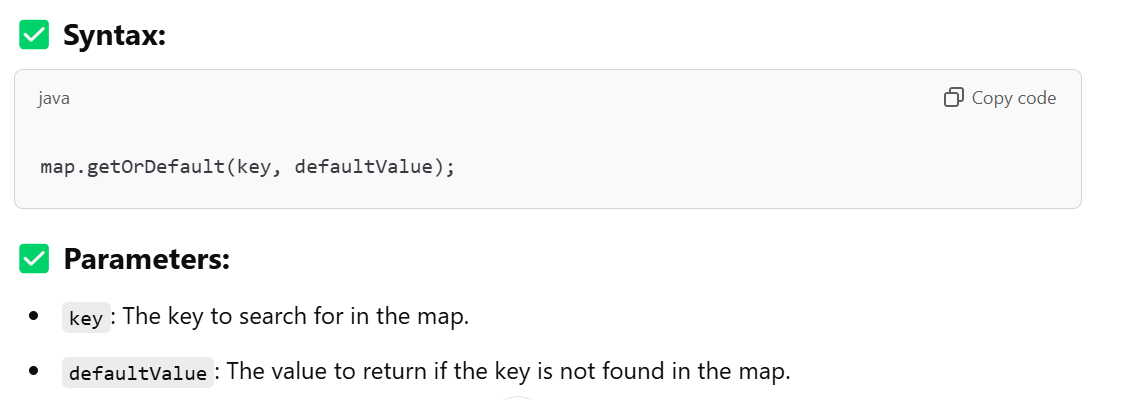
**🔎 Explanation:**

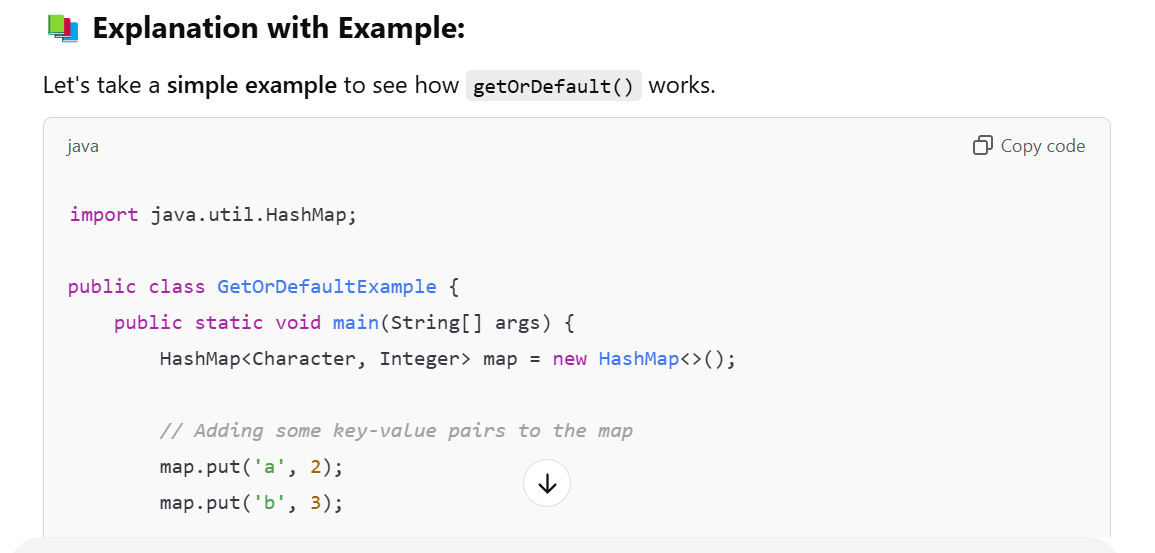
1. **Character Count Logic:**
   * Convert the string to a **character array** using toCharArray().
   * Use a **HashMap** to store each character as a key and its occurrence as a value.
   * Skip spaces while counting.
2. **Word Count Logic:**
   * Split the string into **words** using split(" ").
   * Use a **HashMap** to store each word as a key and its occurrence as a value.

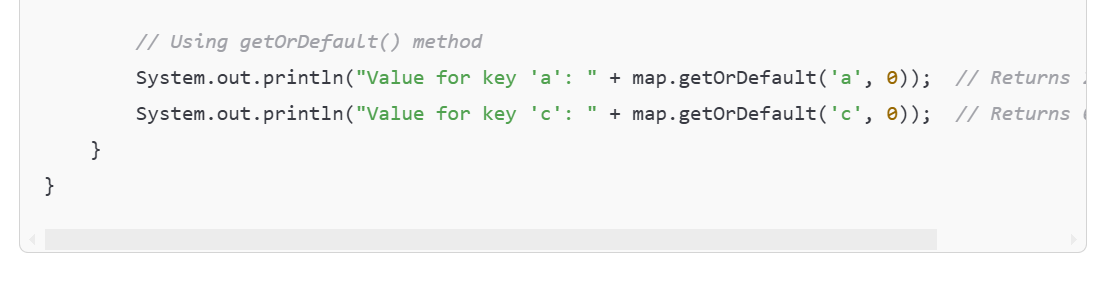


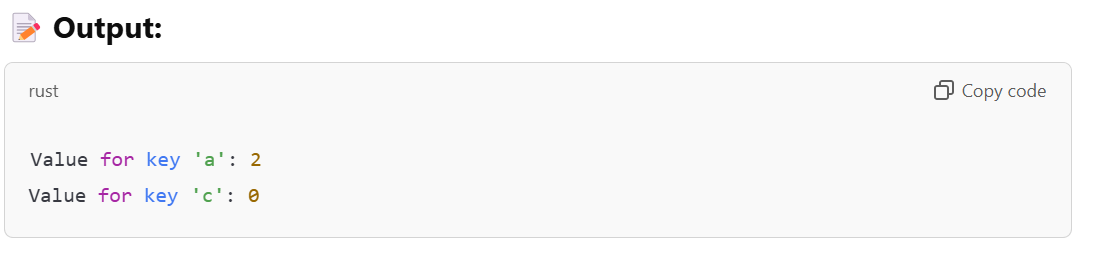
**🔎 What is getOrDefault() Method?**

The **getOrDefault()** method is used with a **Map** (like **HashMap**, **TreeMap**, etc.) to **retrieve the value associated with a given key**. If the key is **not present** in the map, it returns a **default value** that you specify.









**🔧 How It Works:**

1. **map.getOrDefault('a', 0)**:
   * The key **'a'** exists in the map with a value of **2**.
   * So, it returns **2**.
2. **map.getOrDefault('c', 0)**:
   * The key **'c'** does **not exist** in the map.
   * So, it returns the **default value** of **0**.

**🔎 Initial Setup:**

* Input string: **"This is a great thing"**
* **charCount** is an empty HashMap at the start:  
  charCount = {}

**🎯 Dry Run:**

Let's go through each character in the string:

**✅ Iteration 1:**

* **Character:** 'T'
* Check if 'T' is a space:  
  **No**, so we continue.
* **charCount.getOrDefault('T', 0)** returns **0** (since 'T' is not in the map).
* Add 'T' with a count of **1**.

**charCount after this iteration:**

{T=1}

**✅ Iteration 2:**

* **Character:** 'h'
* Check if 'h' is a space:  
  **No**, so we continue.
* **charCount.getOrDefault('h', 0)** returns **0**.
* Add 'h' with a count of **1**.

**charCount after this iteration:**

r

Copy code

{T=1, h=1}

**✅ Iteration 3:**

* **Character:** 'i'
* Check if 'i' is a space:  
  **No**, so we continue.
* **charCount.getOrDefault('i', 0)** returns **0**.
* Add 'i' with a count of **1**.

**charCount after this iteration:**

css

Copy code

{T=1, h=1, i=1}

**✅ Iteration 4:**

* **Character:** 's'
* Check if 's' is a space:  
  **No**, so we continue.
* **charCount.getOrDefault('s', 0)** returns **0**.
* Add 's' with a count of **1**.

**charCount after this iteration:**

css

Copy code

{T=1, h=1, i=1, s=1}

**✅ Iteration 5:**

* **Character:** ' ' (space)
* Check if ' ' is a space:  
  **Yes**, so **skip this character**.

**charCount remains the same:**

css

Copy code

{T=1, h=1, i=1, s=1}

**✅ Iteration 6:**

* **Character:** 'i'
* Check if 'i' is a space:  
  **No**, so we continue.
* **charCount.getOrDefault('i', 0)** returns **1** (since 'i' is already in the map).
* Increment the count of 'i' to **2**.

**charCount after this iteration:**

css

Copy code

{T=1, h=1, i=2, s=1}

**✅ Iteration 7:**

* **Character:** 's'
* Check if 's' is a space:  
  **No**, so we continue.
* **charCount.getOrDefault('s', 0)** returns **1** (since 's' is already in the map).
* Increment the count of 's' to **2**.

**charCount after this iteration:**

css

Copy code

{T=1, h=1, i=2, s=2}

**✅ Iteration 8:**

* **Character:** ' ' (space)
* Check if ' ' is a space:  
  **Yes**, so **skip this character**.

**charCount remains the same:**

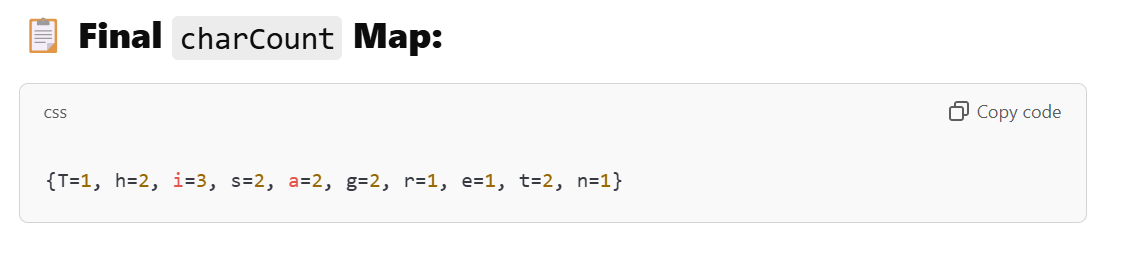
css

Copy code

{T=1, h=1, i=2, s=2}

**✅ Continue with Remaining Characters...**

Following the same logic, here’s how the **charCount** map will be updated after processing all characters:



**🔄 Step 2: Word Count Dry Run**

**🎯 Words Split from String:**

css

Copy code

["This", "is", "a", "great", "thing"]

**🎯 Word Loop:**

**➡️ Iteration 1:**

* **Word:** "This"
* **wordCount:** { "This" -> 1 }

**➡️ Iteration 2:**

* **Word:** "is"
* **wordCount:** { "This" -> 1, "is" -> 1 }

**➡️ Iteration 3:**

* **Word:** "a"
* **wordCount:** { "This" -> 1, "is" -> 1, "a" -> 1 }

**➡️ Iteration 4:**

* **Word:** "great"
* **wordCount:** { "This" -> 1, "is" -> 1, "a" -> 1, "great" -> 1 }

**➡️ Iteration 5:**

* **Word:** "thing"
* **wordCount:** { "This" -> 1, "is" -> 1, "a" -> 1, "great" -> 1, "thing" -> 1 }

