**Collections**: Refer to a group of objects that are stored and manipulated in a structured way. Collections provide a set of classes and interfaces that make it easier to handle groups of objects. The Java Collections Framework (JCF) is a group of classes and interfaces that implement various types of collections. These collections can store elements like data, objects, or even other collections.

The primary goal of collections is to store, manage, and manipulate a group of objects in an efficient and easy-to-use manner.

In Java, both **arrays** and **collections** are used to store multiple elements, but they differ in their characteristics and functionality.

**Array in Java:**

An **array** is a fixed-size data structure that holds a collection of elements of the same type. Once the size of an array is declared, it cannot be changed during runtime.

* **Syntax**:



int[] arr = new int[5]; // An array of 5 integers

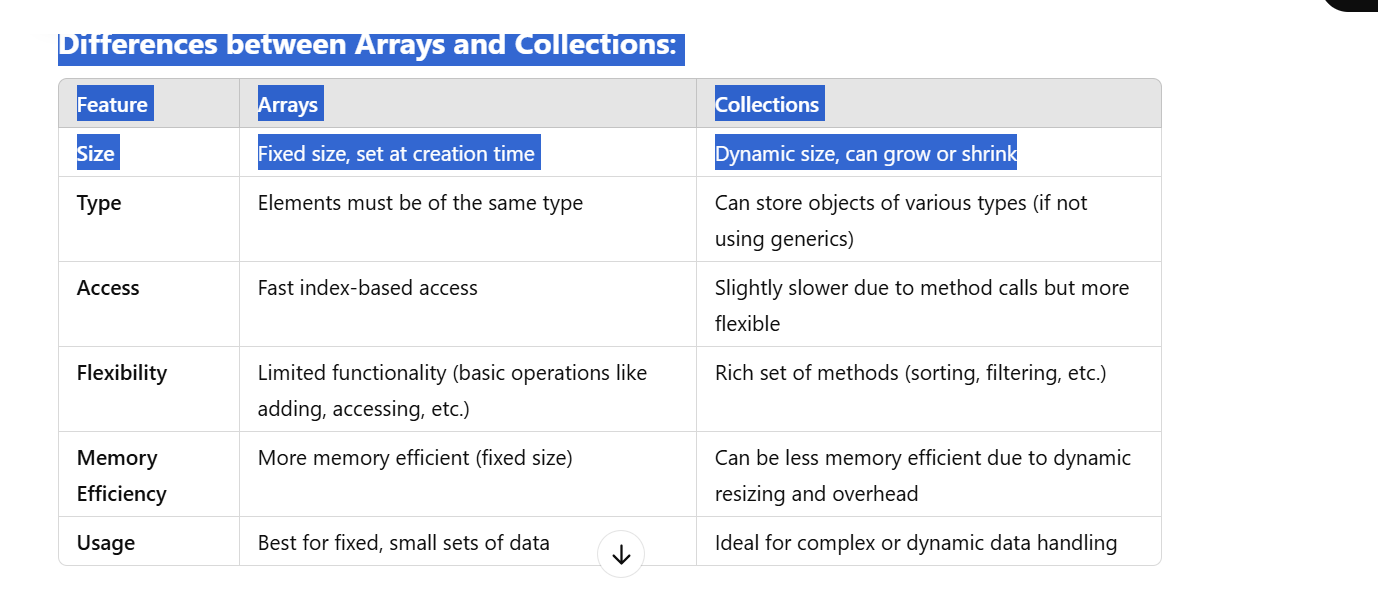
* **Key Characteristics**:
  + **Fixed size**: The size of an array is fixed at the time of creation and cannot be changed later.
  + **Type-specific**: All elements in the array must be of the same type either primitive or object type.
  + **Index-based access**: Array elements are accessed via their index (starting from 0).
  + **Performance**: Arrays offer fast access to elements since it uses contiguous memory locations.

A **collection** is an object that represents a group of objects, but unlike arrays, collections are part of the Java **Collections Framework**, which provides more flexibility and features.

* Collections in Java are part of the java.util package and include interfaces like **List**, **Set**, and **Queue**.
* **Syntax**:

List<Integer> list = new ArrayList<>();

* **Key Characteristics**:
  + **Dynamic size**: Collections can grow and shrink dynamically at runtime.
  + **Type flexibility**: Collections can hold objects of different types depending on the implementation.
  + **More functionality**: Collections provide a rich set of methods for manipulation, searching, sorting, and other operations.
  + **Generics**: Collections use generics to enforce type safety.

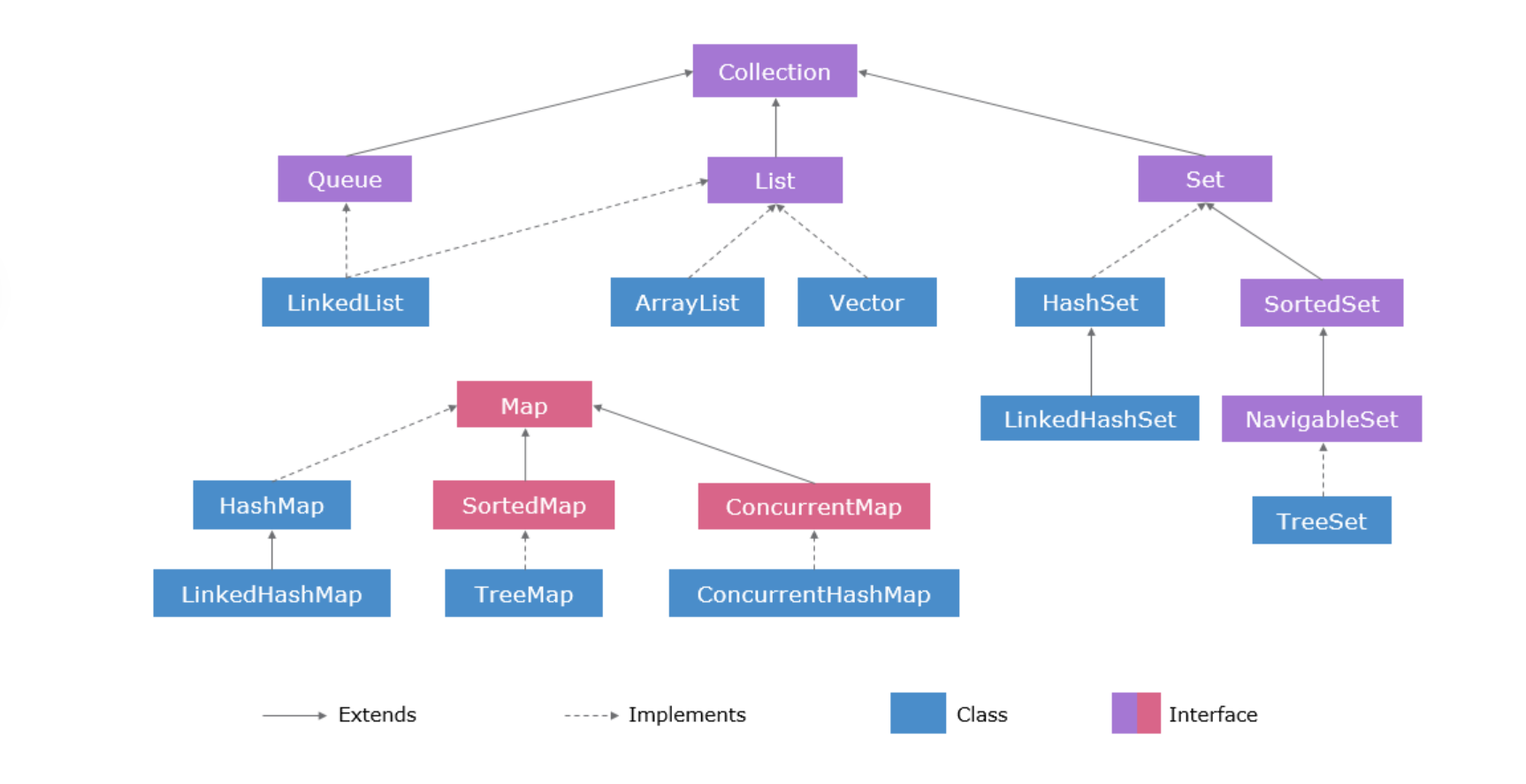


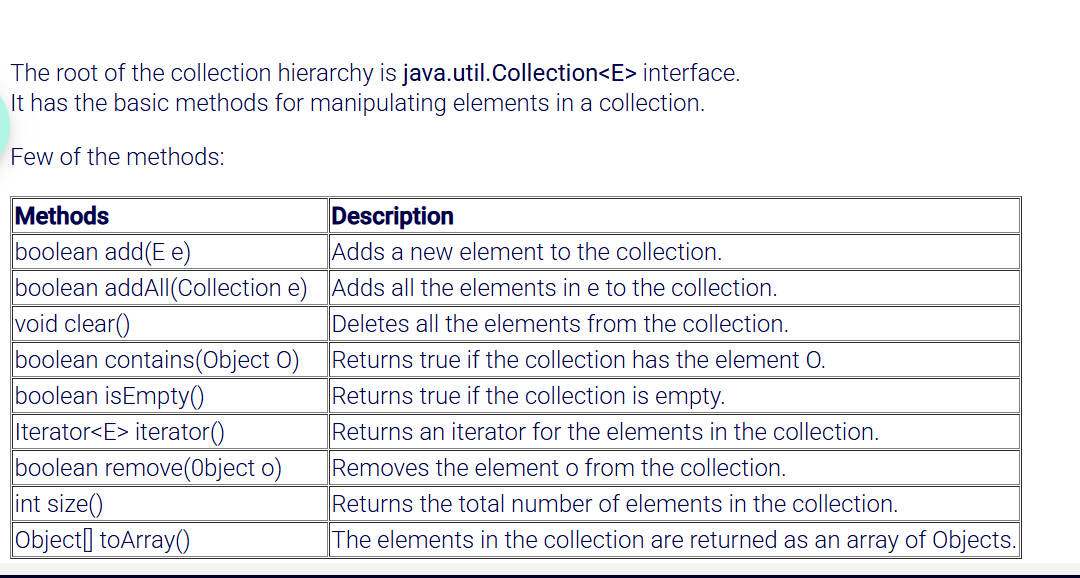
**When to Use Arrays:**

* When the size of the data structure is known and fixed.
* When you need fast access to elements by index.
* When memory efficiency is crucial, and you don’t need extra features.

**When to Use Collections:**

* When the size of the data structure may change dynamically.
* When you need additional functionality, such as sorting, searching, or removing duplicates.
* When working with more complex objects that require flexible management and manipulation.





**Collection** is the root interface of the entire collections framework. It defines the basic operations that can be performed on a collection of objects.

It doesn't provide direct implementations, but other more specific collection interfaces (such as List, Set, and Queue) extend it.

It defines methods for adding, removing, and querying elements, such as add(), remove(), contains(), size(), etc.

**Collections (Utility Class)**

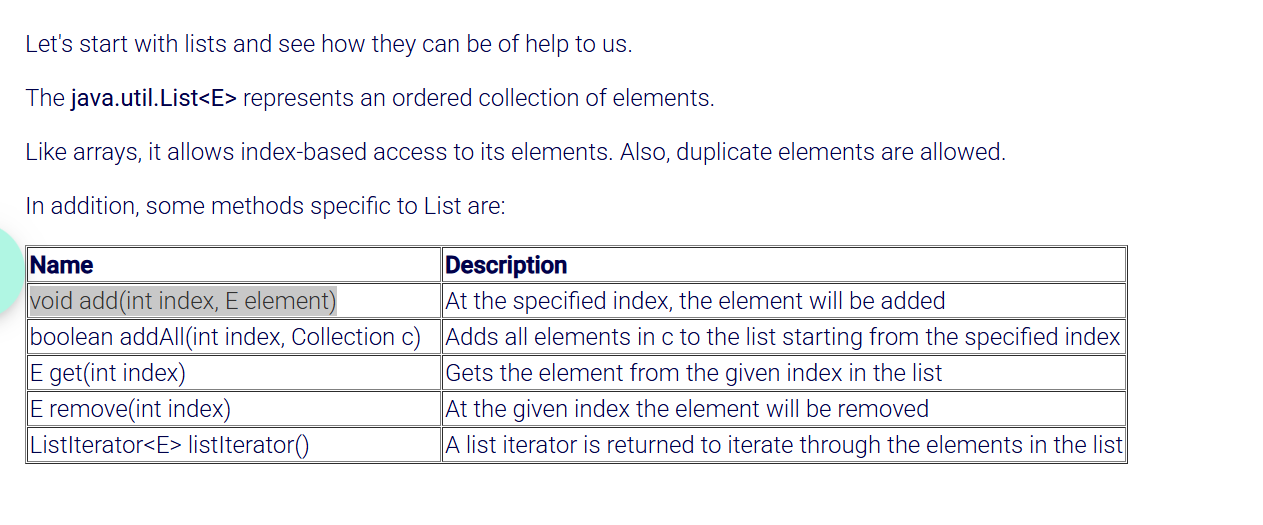
* **Collections** is a **utility class** in Java, meaning it provides **static methods** for manipulating and operating on collections. It is **not an interface** and **does not extend Collection**.
* It is a **helper class** that provides **utility methods** for working with collections.

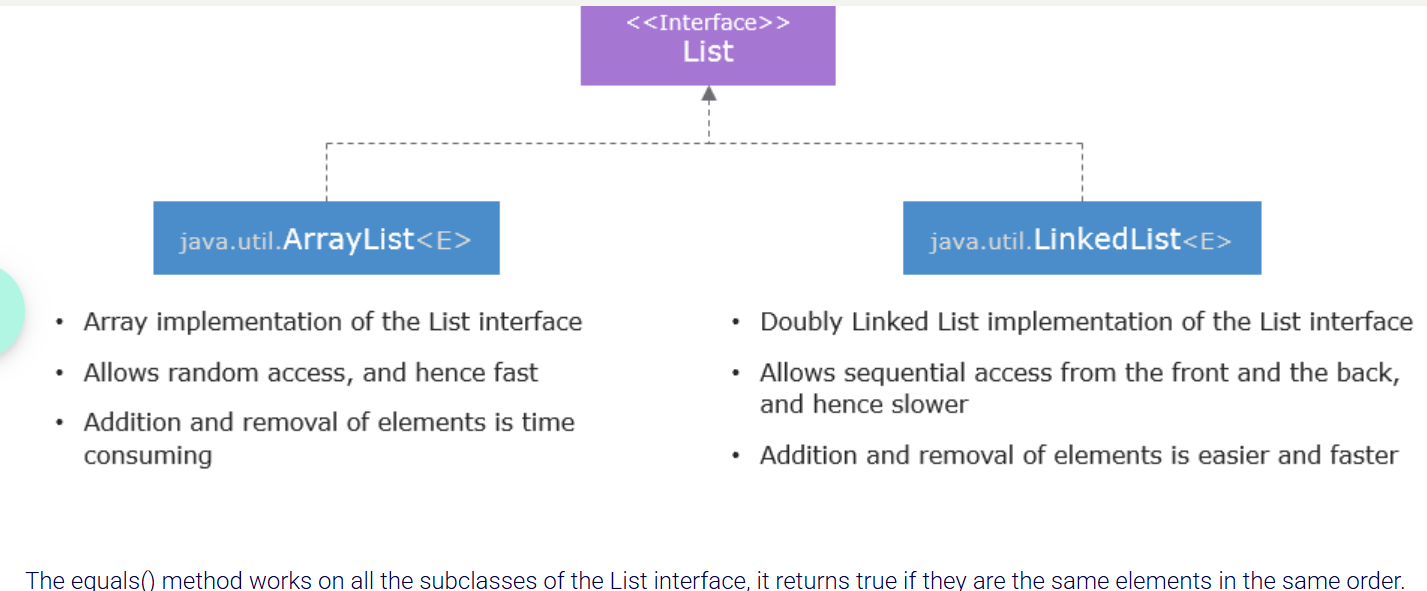
Common methods in Collections include:

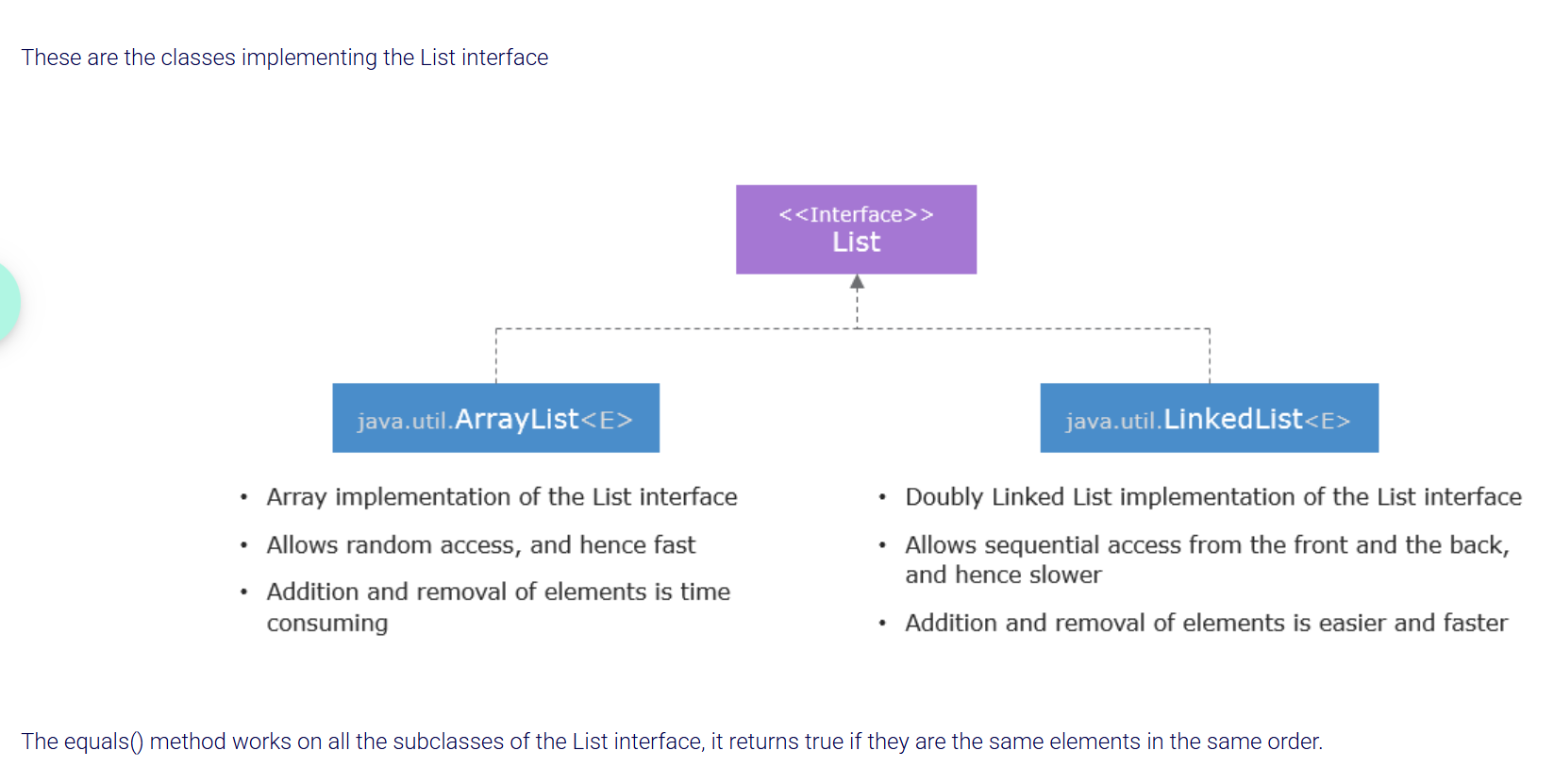
* sort(List<T> list): Sorts a list in natural order.
* shuffle(List<?> list): Randomly shuffles the elements of a list.
* reverse(List<?> list): Reverses the order of elements in a list.
* synchronizedList(List<T> list): Returns a thread-safe version of a list

**Major Types of Collections in Java**

1. **List**:
   * An ordered collection (also known as a sequence). Lists **allow** **duplicate** elements and maintain **the insertion order.**
   * Common Implementations:
     + **ArrayList**: A resizable array implementation of the List interface. It provides fast access to elements by index but can be slow when inserting or deleting elements in the middle.
     + **LinkedList**: A doubly linked list implementation of the List interface. It allows efficient insertions and deletions but slower access by index.







Key methods of the List interface:

* add(E e) – Adds an element to the list.
* get(int index) – Retrieves the element at a specific index.
* remove(int index) – Removes the element at the specified index.
* size() – Returns the size of the list.
* contains(Object o) – Checks if the list contains a particular element.

**Final** keyword usage in list::

A screenshot of a computer

AI-generated content may be incorrect.

**Linked List**: It is a type of **linear data structure** where each element (node) in the list points to the next element, creating a chain-like structure. This allows for efficient insertions and deletions at both ends of the list, especially in comparison to an array-based **ArrayList.**

Unlike arrays, where the memory is contiguous (i.e., each element is stored in a consecutive memory block), a linked list consists of nodes that are dynamically linked together, where each node contains data and a reference (or link) to the next node.

**Key Features of LinkedList:**

1. **Node-based Structure**: A LinkedList is made up of nodes, where each node stores:
   * **Data**: The actual value or data of the node.
   * **Next Reference**: A reference to the next node in the list.

In a **doubly linked list** (which is the implementation of LinkedList in Java), each node also has a reference to the previous node.

**Types of LinkedLists in Java:**

* **Singly Linked List**: Each node points to the next node.
* **Doubly Linked List**: Each node has two references—one to the next node and one to the previous node. Java’s LinkedList class is based on this type.

A **node** in a **LinkedList** is simply a **container** that holds two pieces of information: Data and Reference(doubly linked list has two references).

**Example of a Doubly LinkedList:**

Let’s say we have a **Doubly LinkedList** with 3 elements: 10 <-> 20 <-> 30.

* The **first node** will hold:
  + data = 10
  + next = reference to the second node
  + prev = null (since it’s the first node, there’s no previous node)
* The **second node** will hold:
  + data = 20
  + next = reference to the third node
  + prev = reference to the first node
* The **third node** will hold:
  + data = 30
  + next = null (since it’s the last node)
  + prev = reference to the second node

**Common Operations Supported by LinkedList:**

1. **Add Elements**:
   * add(E element): Adds an element at the end of the list.
   * addFirst(E element): Adds an element at the beginning of the list.
   * addLast(E element): Adds an element at the end of the list (same as add()).
   * add(index, E element): Adds an element at a specified index.
2. **Remove Elements**:
   * remove(): Removes and returns the first element.
   * removeFirst(): Removes the first element.
   * removeLast(): Removes the last element.
   * remove(index): Removes the element at the specified index.
3. **Access Elements**:
   * get(index): Retrieves the element at the specified index.
   * getFirst(): Retrieves the first element.
   * getLast(): Retrieves the last element.
4. **Other Useful Methods**:
   * size(): Returns the number of elements in the list.
   * isEmpty(): Checks if the list is empty.
   * contains(Object o): Checks if a specific element is in the list.
   * clear(): Removes all elements from the list.

**Efficient Insertion/Deletion**: Insertion and deletion operations are **efficient** (O(1) time complexity) at both the beginning and end of the list, because there is no need to shift elements as in an array-based list.

 **Slower Access**: **Random access** (accessing an element at a particular index) is slower in a LinkedList compared to an ArrayList because you must traverse the list from the beginning (or end) to reach the desired index, resulting in a time complexity of O(n).

 **Supports Null Values**: Both ArrayList and LinkedList can store **null** elements.

What is the difference between ArrayList and LinkedList in Java?

**ArrayList** and **LinkedList** are both **List** implementations in Java, but they have different internal structures and performance characteristics.

**ArrayList** : Elements are stored in contiguous memory locations.

**LinkedList** : Each element (node) stores the data and references (or links) to the previous and next node. The nodes are not stored in contiguous memory locations.

**2. Access Time (Indexing):**

 **ArrayList**:

* **O(1)** time for accessing elements by index. Since it's backed by an array, accessing any element by its index is fast and constant time.

 **LinkedList**:

* **O(n)** time for accessing elements by index. To access an element, the list needs to be traversed from the head to the desired position (or from the tail if closer) but its less efficient and performance can degrade because get(index) will have to traverse the nodes one by one.

**3. Insertion and Deletion:**

* **ArrayList**:
  + **Adding elements** at the end is generally **O(1)**, unless the array needs to be resized.
  + **Inserting or deleting elements** in the middle or beginning is **O(n)** because all subsequent elements must be shifted to accommodate the change.
* **LinkedList**:
  + **Insertion and deletion** at the beginning or middle is **O(1)** (if you have a reference to the node). Since it’s a linked list, nodes can be added or removed by adjusting a few references without shifting other elements.
  + **Adding at the end** is **O(1)** if you maintain a reference to the last node. However, if you don’t maintain a reference to the last node, it would take **O(n)** to traverse the list and add the new node at the end.
* be traversed from the head to the desired position (or from the tail if closer).

**4. Memory Usage:**

* **ArrayList**:
  + Since **ArrayList** uses a contiguous block of memory, it requires less memory overhead.
  + The array has to be resized occasionally as elements are added, which involves copying the entire array to a new larger array.
* **LinkedList**:
  + Each node in a **LinkedList** requires additional memory for storing references to the next and previous nodes (in the case of a doubly linked list).
  + So, **LinkedList** generally uses more memory than **ArrayList**.

How do we sort a list in java?

**Using Collections.sort() (Natural Ordering)**

If the elements in the list implement the Comparable interface (like Integer, String, etc which implements the Comparable interface by **default**), you can use Collections.sort() to sort the list in its **natural order** (ascending order for **numbers** or alphabetical order for **strings** or whatever ordering is defined in compareTo() method for **custom class**).



A white background with text

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

For **custom** objects, your class must implement **Comparable** and override the **compareTo**() method to define the natural order(either asc or desc). it can be sorted using Collections.sort() or List.sort() without needing an external Comparator.

Collections.reverseOrder() works for custom classes too.

 Collections.sort() Works only on **Lists** (e.g., ArrayList, LinkedList). On the other hand, **Sets** (like HashSet, TreeSet) are not ordered by definition (except for **TreeSet**, which keeps elements sorted but does not use the Collections.sort() method).

 The elements of the list must implement **Comparable** (i.e., be naturally sortable).

 It **modifies the original list** (in-place sort).

 For custom sorting (e.g., descending), you can use:

Collections.sort(list, Collections.reverseOrder());

Collections.sort(list) only works **if the elements in the list are *Comparable*** — meaning, Java must know **how** to compare two objects in the list to determine their order. For **integers** and **Strings** they already know coz these classes implement comparable by default and no need to do any explicit comparison and can sort directly using **Collections.sort();**

Collections.sort(list) works directly for any class(custom classes too) that implements Comparable.

You **can** implement a custom order using Comparable by defining the sorting logic inside compareTo(). However, it’s **fixed** and less flexible than Comparator.

You can sort **any data type** (int, double, String, etc.) using Comparable, and you can choose to sort in either **ascending or descending order** — *but only one of those* at a time. That becomes the "natural" or default ordering.

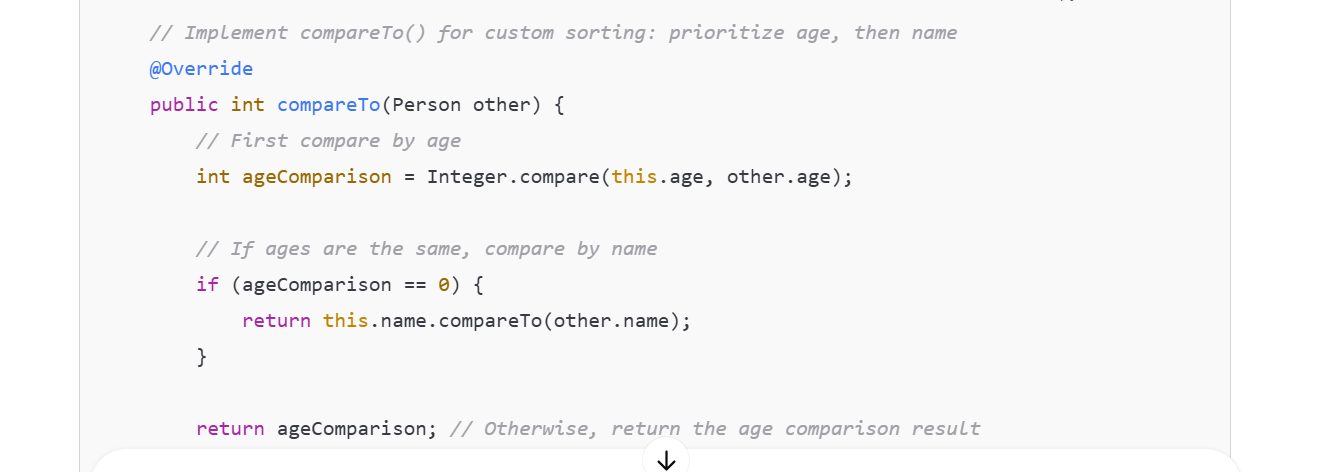
**✅ You *can* sort by:**

* int, double → using Integer.compare, Double.compare
* String → using .compareTo() (since String implements Comparable)
* Any class that implements Comparable

Comparable only lets you define **one default sorting logic** inside your class. If you want to sort by different fields at different times, use Comparator.

You can't easily switch behavior using just Comparable. That's where Comparator comes in.

In below eg: Person class implementing Comparable



**Comparable** :

In Java, Comparable is an **interface** that a class can implement to define a **natural ordering** for its objects. This interface allows objects to be compared to each other, which is necessary for operations like sorting etc on lists,sets.

If a class implements the **Comparable** interface, it must override the **compareTo**() method, which compares the current object (this) with another object of the same type.

Built-in classes like String, Integer, and Double **already implement Comparable**.

So when you do:

List<Integer> list = Arrays.asList(5, 2, 8);

Collections.sort(list); //[2,5,8]

It works because Integer already knows how to compare itself.

Here's the basic structure of the **Comparable** interface:

public interface Comparable<T> {

public int compareTo(T o);

}

The **compareTo**() method is the key method in the Comparable interface. It returns an integer:

* A negative integer if the current object is less than the object passed as the argument.
* Zero if the current object is equal to the object passed as the argument.
* A positive integer if the current object is greater than the object passed as the argument.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

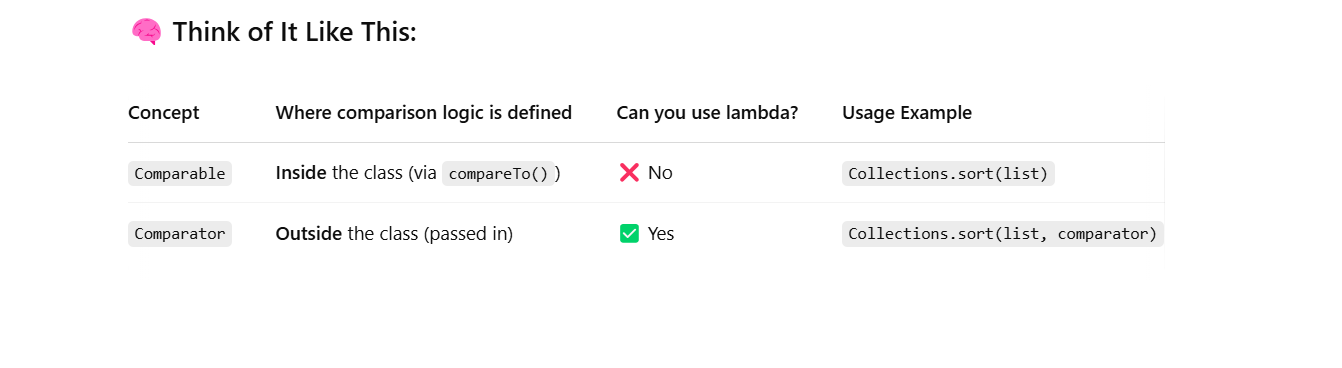
AI-generated content may be incorrect.

**Why Use Comparable?**

* **Sorting**: It allows objects to be sorted naturally (e.g., alphabetically, numerically).
* **Consistent comparisons**: It guarantees consistent comparisons between objects, which is important for algorithms like sorting, binary search, etc.

**Key Points:**

* A class can only implement one **compareTo**() method (as there's only one natural order), so you can't have multiple ways to compare the objects using Comparable.
* If you want to compare objects in multiple ways (e.g., by age and name), you can use the Comparator interface, which allows multiple comparison strategies.Can compare objects by age then name in comparable as well buts it’s a single sorting. If we want to sort byname then age ..we would need to modify compareto method according to this…so only one sorting rule is allowed for compareTo. Since we cant have multiple compareTo() methods inside a class implementing comparable.
* Collections.sort(list) is to sort a list of objects, but you need to ensure that the objects can be compared in some way to determine the order (either by implementing Comparable(so internally or providing a Comparator).
* The compareTo() method is invoked whenever the Java runtime needs to compare two objects of the class that implements Comparable Interface.
* We cant write lambda expression to implement comparable interface.
* Comparable is meant to be **implemented directly by the class**(inside pojo class) whose objects need a "natural order." That means you define the compareTo() method *inside your class*. You can’t apply a lambda expression directly to change or inject Comparable logic outside of the class — that's what **Comparator** is for.
* The reason Comparable needs to be implemented inside the class is because its compareTo() method takes **one parameter** and uses this as the other object.



 With \*\*Comparable\*\*, the sorting logic lives **inside the class being sorted** (like Person implements Comparable<Person> { ... })

✅ That defines the **natural/default** way to compare Person objects — and it's *baked into* the class.

 With \*\*Comparator\*\*, you write the sorting logic **outside** the Person class — either as:

* A separate class (like AgeComparator implements Comparator)
* An anonymous class
* A lambda expression

**Comparator** :

A **Comparator** in Java is an interface used to define a custom ordering (or sorting logic) for objects that are not naturally comparable or when you want a different ordering than the one provided by the natural ordering (i.e., the compareTo() method in Comparable).

This is particularly useful when you want to sort objects in multiple ways (e.g., by age, name, etc.) without changing the class's compareTo() implementation.

And used when we don’t want to modify the class because **compareTo**() accepts only one argument, so we need the calling object (this) to compare with it — that's why we implement Comparable *inside* the class.

**compare**() takes two arguments(2 objects), so it doesn’t need to be inside the class."

public interface Comparator<T> {

int compare(T o1, T o2);

}

**Why Use a Comparator?**

* When you want to **sort a collection** based on a custom ordering that isn’t the natural ordering (i.e., using compareTo()).
* When you need to compare objects based on multiple criteria or different fields.

**Using Lambda Expression:**

A screenshot of a computer code

AI-generated content may be incorrect.

A close-up of a computer screen

AI-generated content may be incorrect.

**Using Comparator Interface :**

* This is an **anonymous inner class** — a quick, inline way to create an object that implements Comparator<Person> **without creating a separate named class**.



**Using separate classes that implements comparator :**

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

**🔑 So You're Right:**

* **compareTo() needs this**, so it must be implemented inside the class.
* **compare() gets both objects**, so it can be implemented **outside** the class.
* The reason Comparator can be implemented in a separate class is because its compare() method takes **two parameters**, so you don't need to be inside either object to compare them.

**ITERATOR:**

An **Iterator** is an interface in the java.util package that allows you to iterate over collections like lists, sets, and maps in a uniform manner. It provides methods for traversing through the elements of a collection, removing elements from the collection during iteration, and handling the iteration process without exposing the underlying structure.

Iterator is an interface used to traverse or iterate over a collection(List,Set,Queue) and access the elements one by one.

 An iterator allows you to access each element in the collection **sequentially**.

 You can **get** the element, **check** if more elements are available, and **remove** elements (in some cases).

Get – next()

Check – hasNext()

Remove – remove()

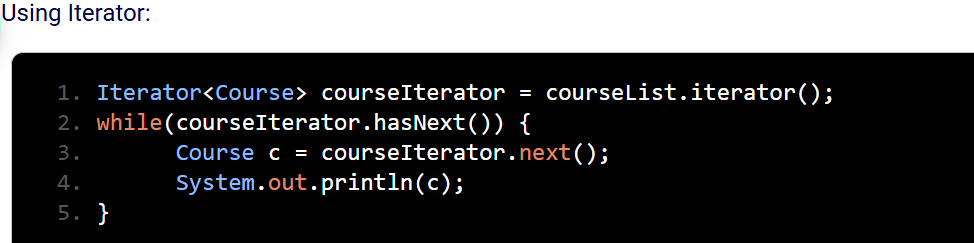
**Iterators** play a crucial role in safely and efficiently traversing through collections. They provide a uniform way to access elements in any collection and allow for operations like removal during iteration, ensuring that you avoid common pitfalls like concurrent modification errors.

A screenshot of a computer

AI-generated content may be incorrect.

**Key Methods of the Iterator Interface:**

1. boolean **hasNext()**: Returns true if there are more elements to iterate over.
2. **next()**: Returns the next element in the iteration. Throws NoSuchElementException if no more elements are present.
3. void **remove()**: Removes the last element returned by the iterator (optional operation).Not supported with this kind of list declaration (like Arrays.asList())



**Iterator vs. Enhanced for-loop (for-each loop):**

While you can use the **enhanced for-loop** (for-each loop) in Java to iterate over collections, it does not provide direct access to the Iterator methods like remove() and cannot check if elements are available further in the list.

For example, the enhanced for-loop works fine for reading elements, but you cannot modify the collection using it:

// Using enhanced for-loop

for (String item : list) {

System.out.println(item); // Cannot call remove() here

}

A **ListIterator** is a special type of iterator used to iterate through a **List** in Java, which is part of the java.util package. It provides functionality to traverse the list in both **forward** and **backward** directions, and allows you to **modify** the list during iteration.

Here are the key features of a ListIterator:

* **Bidirectional iteration**: Unlike a regular Iterator, a ListIterator allows iteration in both directions (forward and backward).
* **Modification**: It allows you to modify the list while iterating, using methods like add(), remove(), and set().
* **Access to indices**: You can also get the current position in the list.

**Common Methods of ListIterator:**

1. **hasNext()**: Returns true if there is another element when iterating forward.
2. **next()**: Returns the next element in the list and moves the cursor forward.
3. **hasPrevious()**: Returns true if there is a previous element when iterating backward.
4. **previous()**: Returns the previous element in the list and moves the cursor backward.
5. **add(E element)**: Adds the specified element to the list at the current position (or between the previous and next elements).
6. **remove()**: Removes the last element returned by next() or previous().
7. **set(E element)**: Replaces the last element returned by next() or previous() with the specified element.
8. **nextIndex()**: Returns the index of the next element in the list (or the size if at the end).
9. **previousIndex()**: Returns the index of the previous element in the list (or -1 if at the start).

**Example of Using ListIterator:**

import java.util.\*;

public class ListIteratorExample {

public static void main(String[] args) {

// Create a List of Strings

List<String> list = new ArrayList<>(Arrays.asList("Apple", "Banana", "Cherry", "Date"));

// Get a ListIterator for the list

ListIterator<String> listIterator = list.listIterator();

// Iterate forward through the list

System.out.println("Iterating forward:");

while (listIterator.hasNext()) {

System.out.println(listIterator.next());

}

// Iterate backward through the list

System.out.println("\nIterating backward:");

while (listIterator.hasPrevious()) {

System.out.println(listIterator.previous());

}

// Modify the list during iteration

listIterator = list.listIterator(); // Reset the iterator to the start

while (listIterator.hasNext()) {

String current = listIterator.next();

if (current.equals("Banana")) {

listIterator.set("Blueberry"); // Replace "Banana" with "Blueberry"

}

}

// Display the modified list

System.out.println("\nModified list:");

System.out.println(list);

}

}

**Output:**

Iterating forward:

Apple

Banana

Cherry

Date

Iterating backward:

Date

Cherry

Banana

Apple

Modified list:

[Apple, Blueberry, Cherry, Date]

**Explanation:**

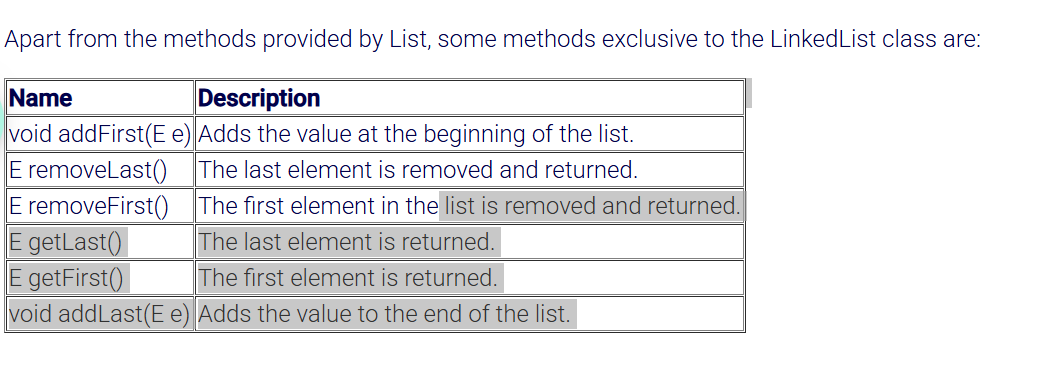
1. The **forward iteration** goes from the first to the last element, using hasNext() and next().
2. The **backward iteration** goes from the last to the first element, using hasPrevious() and previous().
3. During iteration, we use set() to modify an element (replacing "Banana" with "Blueberry").

**When to Use ListIterator:**

* When you need to iterate in both directions (forward and backward).
* When you need to modify the list while iterating (adding(**add**()), removing(**remove**()), or updating elements(**set**()).
* When working with lists where you need more control over iteration, such as getting the current index or using the iterator to both read and modify elements.

**Iterator**:

* Allows removing elements(**remove**()).
* Does **not support adding or modifying** elements directly.



The Edford data management team says that the courses are unique. Using lists to hold them can allow duplicate values, which is not at all desired.

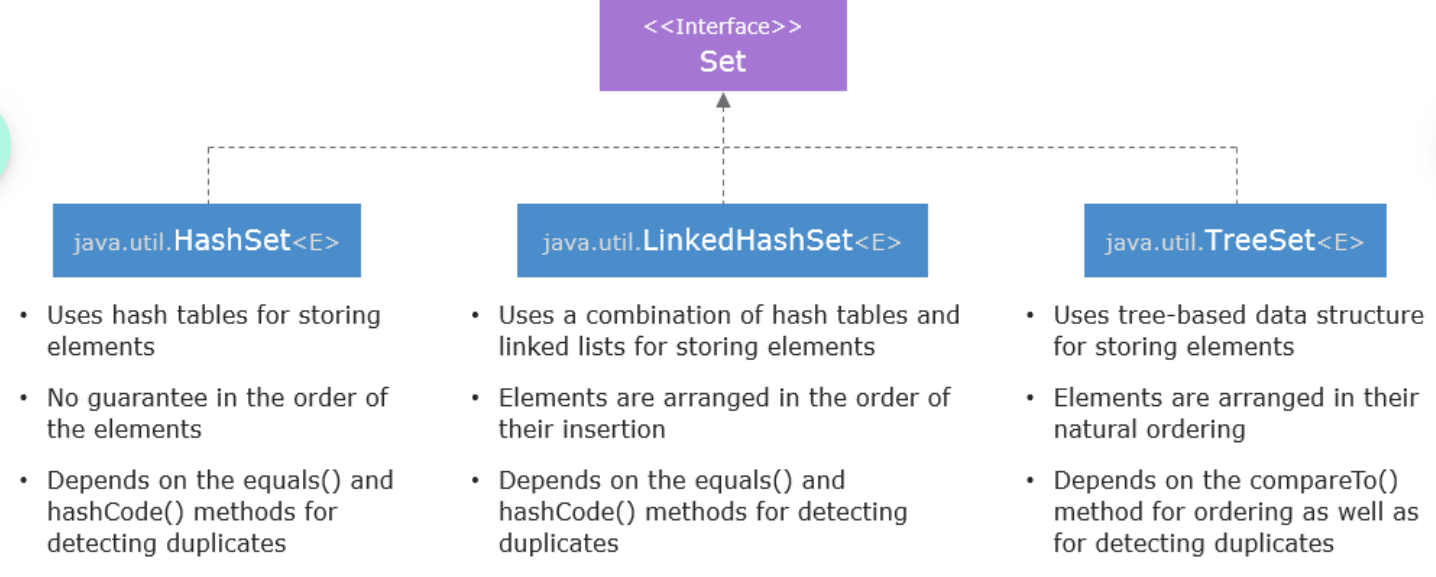
To solve this, we can use **sets**.

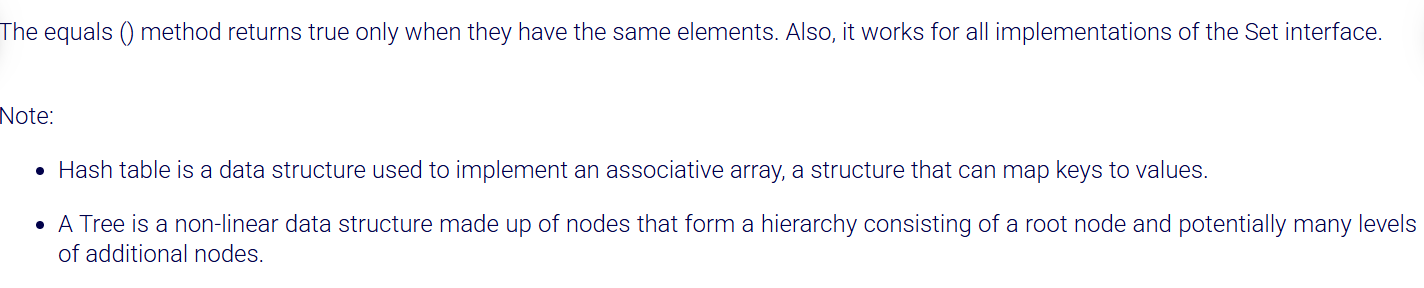
The**java.util.Set<E>** represents a collection of unique items, i.e. duplicate data is not allowed.

Set has only the methods from the Collection interface.  
Since sets are unordered, they can not be accessed using indexes. The enhanced for loop and iterator helps to access the elements of a set.

Let's have a look at some classes implementing the Set interface.

Note: The add() method will return false if our program attempts to add a duplicate element.





public class setTester {

public static void main(String[] args) {

Set<Integer> setInt = new HashSet<>();

//Since sets are unordered, they can not be accessed using indexes.

//The enhanced for loop and iterator helps to access the elements of a set.

setInt.add(10);

setInt.add(5);

setInt.add(5);

setInt.add(20);

System.***out***.println(setInt); //prints unordered and removes duplicates if any.

//using for each loop

for(Integer i : setInt) {

System.***out***.println(i);

System.***out***.println();

}

//Iterating using iterator

Iterator<Integer> it = setInt.iterator();

while(it.hasNext()) {

int m = it.next();

System.***out***.println(m);

}

// Creating a new TreeSet object

Set<String> courseSet = new TreeSet<>();

// Adding elements to the Set

courseSet.add("Java");

courseSet.add("Hibernate");

courseSet.add("Angular JS");

courseSet.add("Java");

// Iterating over the set using enhanced for loop

for(String s: courseSet) {

System.***out***.println(s); //sorts the elements in asc and removes duplicates if any.

}

// Creating a new LinkedHashSet object

Set<Integer> numberSet = new LinkedHashSet<>();

// Adding elements to the set

numberSet.add(13);

numberSet.add(12);

numberSet.add(24);

numberSet.add(13);

// Displaying the Set

System.***out***.println(numberSet); //prints in insertion order and removes duplicates if any.

System.***out***.println(numberSet.size()); //removes duplicates and prints the size of unique elements

}

}

In Java, a **Set** is a **collection** that does not allow duplicate elements. It is part of the Java Collections Framework and is a commonly used interface to represent a group of elements where each element can appear only once.

**Key Characteristics of a Set:**

* **No Duplicates**: A Set automatically ensures that no duplicate elements are stored. If you attempt to add a duplicate element, it will not be added to the set.
* **Unordered**: The elements in a Set are not ordered. The order in which the elements are added is not necessarily the order in which they are iterated over. However, there are different types of sets that can behave differently regarding order (e.g., LinkedHashSet preserves the insertion order, while TreeSet sorts elements).

**Common Implementations of the Set Interface:**

1. **HashSet**:
   * The most commonly used implementation of the Set interface.
   * It does not guarantee any specific order of elements.
   * It uses a hash table to store the elements, making it very fast for adding, removing, and checking if an element exists.

Set<String> hashSet = new HashSet<>();

hashSet.add("Apple");

hashSet.add("Banana");

hashSet.add("Apple"); // Duplicate, will not be added

System.out.println(hashSet); // Output: [Apple, Banana]

1. **LinkedHashSet**:
   * This implementation of Set maintains the **insertion order** of elements.
   * It behaves like HashSet but with the added feature that the elements are returned in the order they were inserted.

Set<String> linkedHashSet = new LinkedHashSet<>();

linkedHashSet.add("Apple");

linkedHashSet.add("Banana");

linkedHashSet.add("Apple"); // Duplicate, will not be added

System.out.println(linkedHashSet); // Output: [Apple, Banana]

1. **TreeSet**:
   * A TreeSet is a Set that is **sorted** based on the natural ordering of its elements, or by a comparator provided at the time of creation.
   * It does not allow null elements.

Set<Integer> treeSet = new TreeSet<>();

treeSet.add(3);

treeSet.add(1);

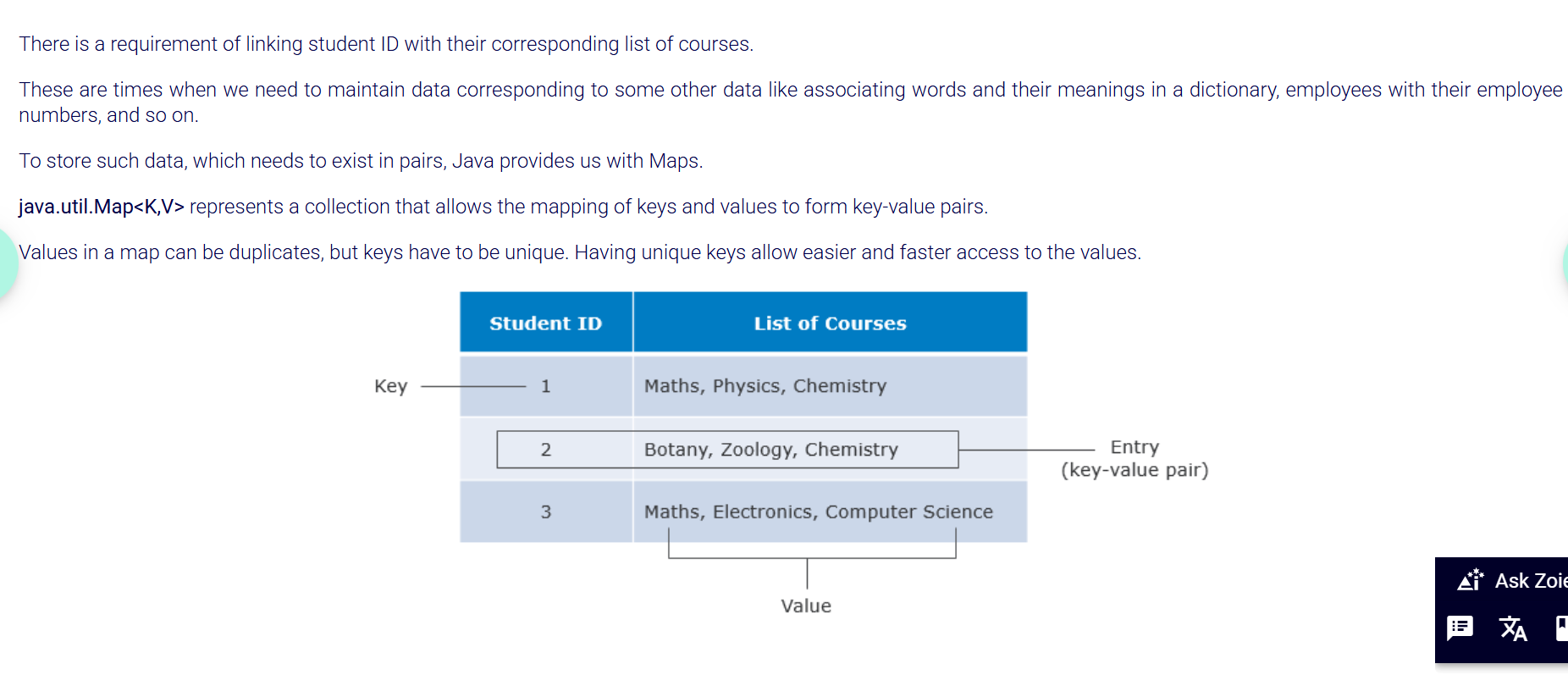
treeSet.add(2);

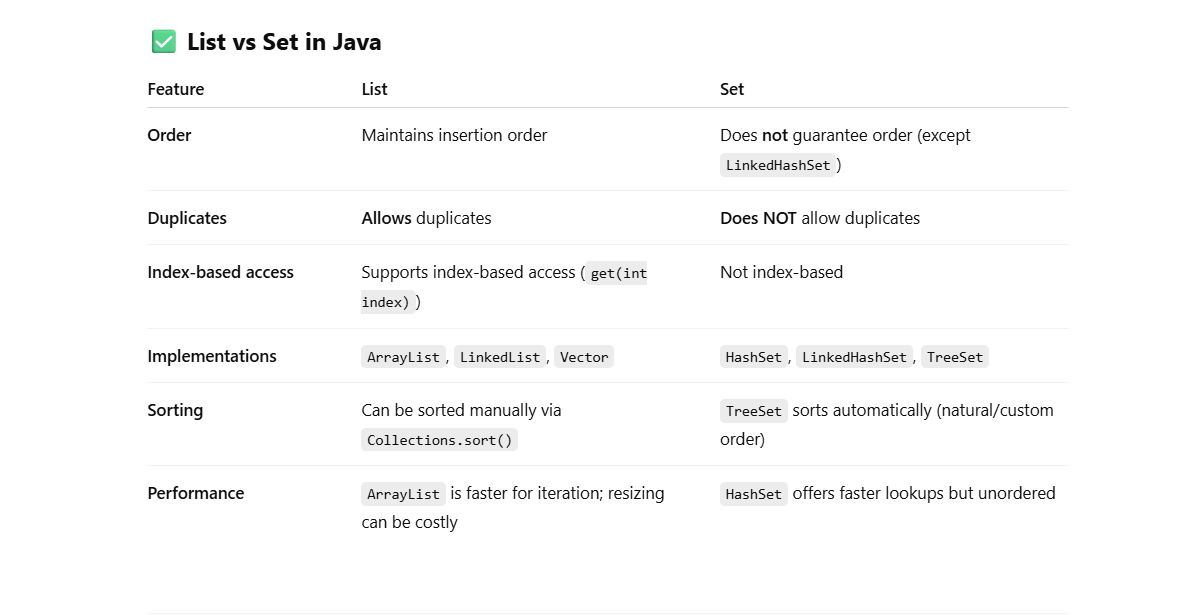
treeSet.add(1); // Duplicate, will not be added

System.out.println(treeSet); // Output: [1, 2, 3]

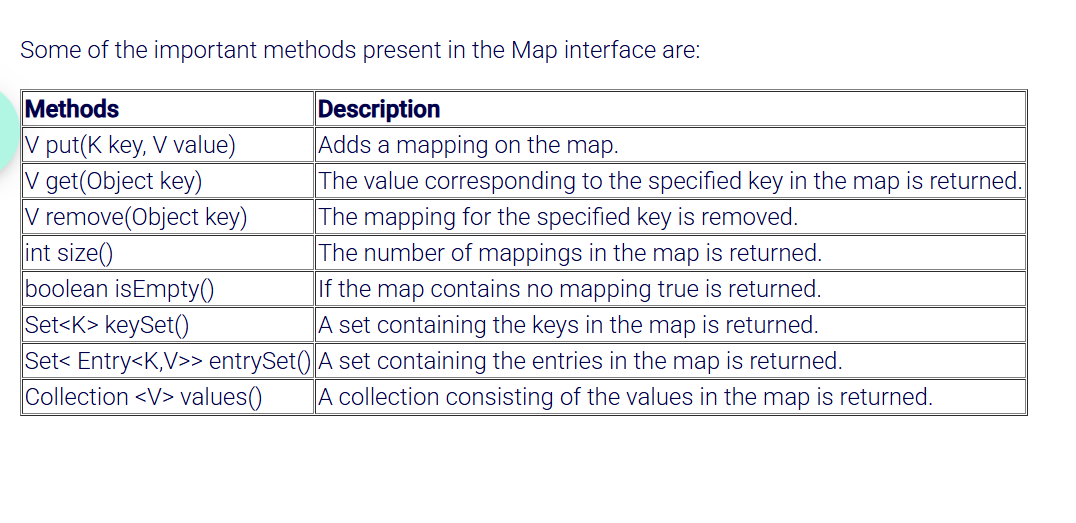
**Operations with Sets:**

* **add(E e)**: Adds the element e to the set, returns true if the set did not already contain the element.
* **remove(Object o)**: Removes the element o from the set, returns true if the set contained the element.
* **contains(Object o)**: Checks if the set contains the element o.
* **size()**: Returns the number of elements in the set.
* **isEmpty()**: Checks if the set is empty.
* **clear()**: Removes all elements from the set.





**Map:**



The **Map** interface does not extend the Collection interface. Hence, there is no iterator for maps. Also, map values cannot be accessed without keys. So a map cannot be traversed directly.

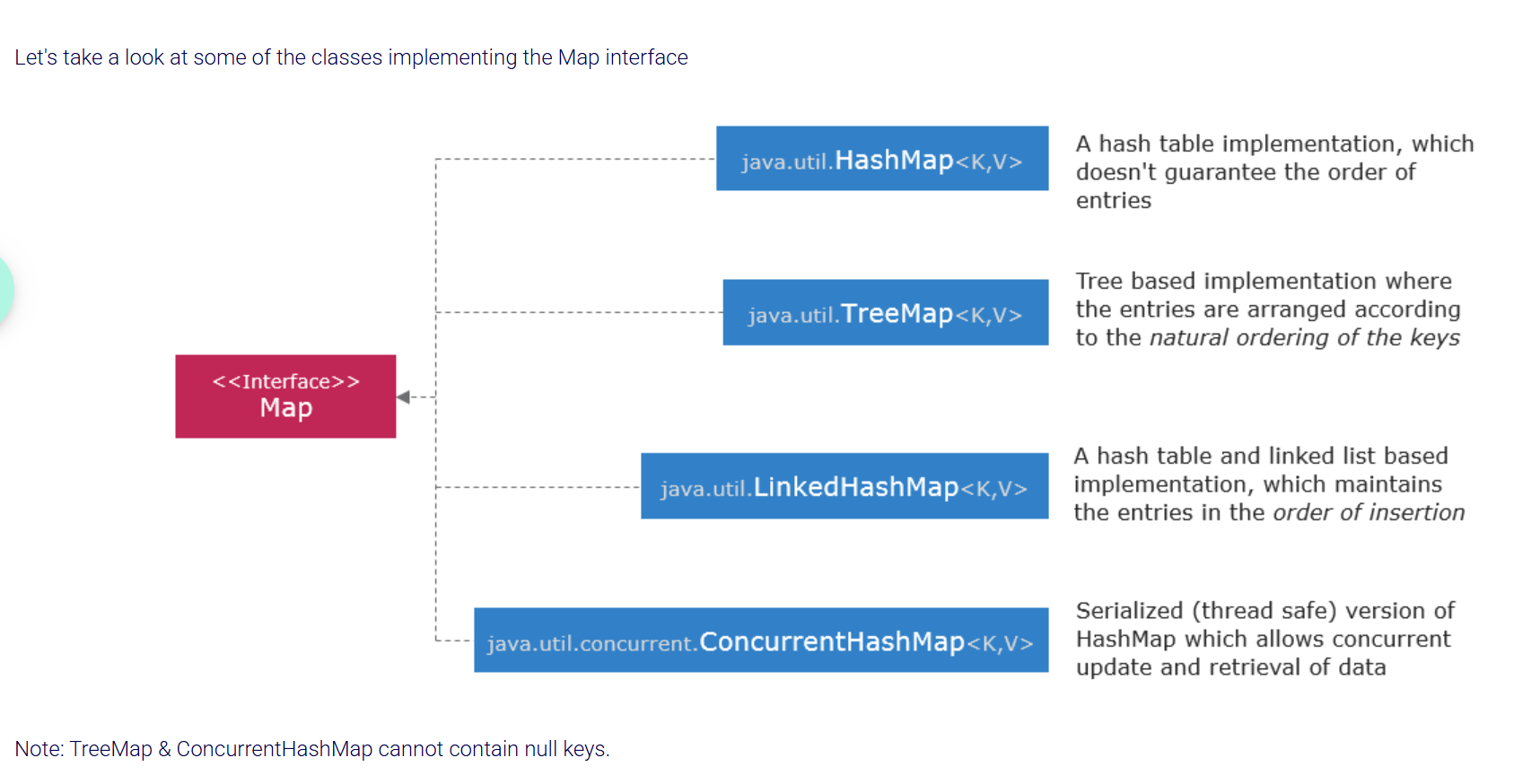
To overcome this, Map provides us with methods to retrieve a Collection that we can traverse.  
There are three different approaches to this:

* Working with the keys:
  1. Set<> setOfKeys = map.keySet();

* Get the Collection of values:
  1. Collection valueCollection = map.values();

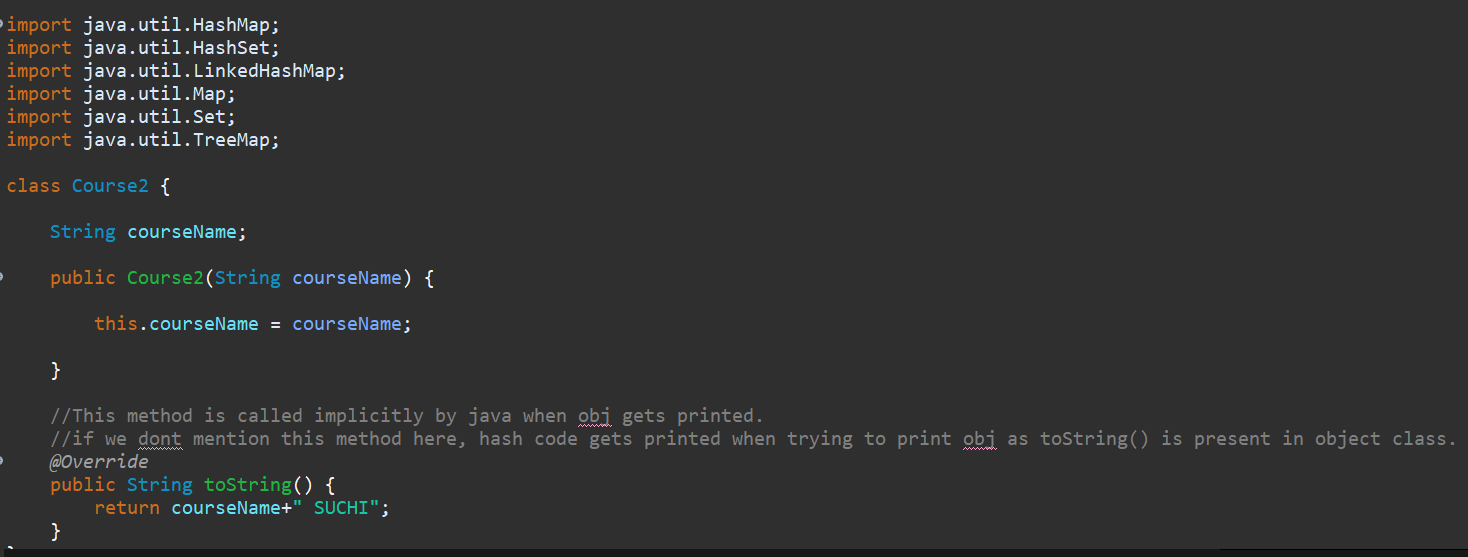
A screen shot of a computer program

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.



A screen shot of a computer program

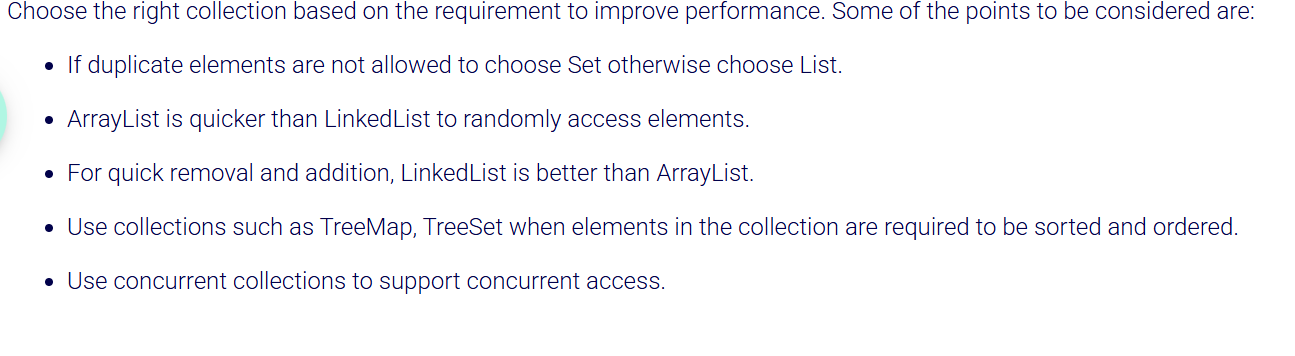
AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

A screen shot of a computer program

AI-generated content may be incorrect.



3️⃣ What is the difference between HashMap and TreeMap?

**1. Implementation**

* **HashMap**: Implements the Map interface and uses a hashtable(An array of buckets) to store data. It does not maintain any order for the entries.
* **TreeMap**: Implements the Map interface and stores its keys in a **red-black tree** (a self-balancing binary search tree). It maintains the keys in a **sorted order**.

**2. Ordering**

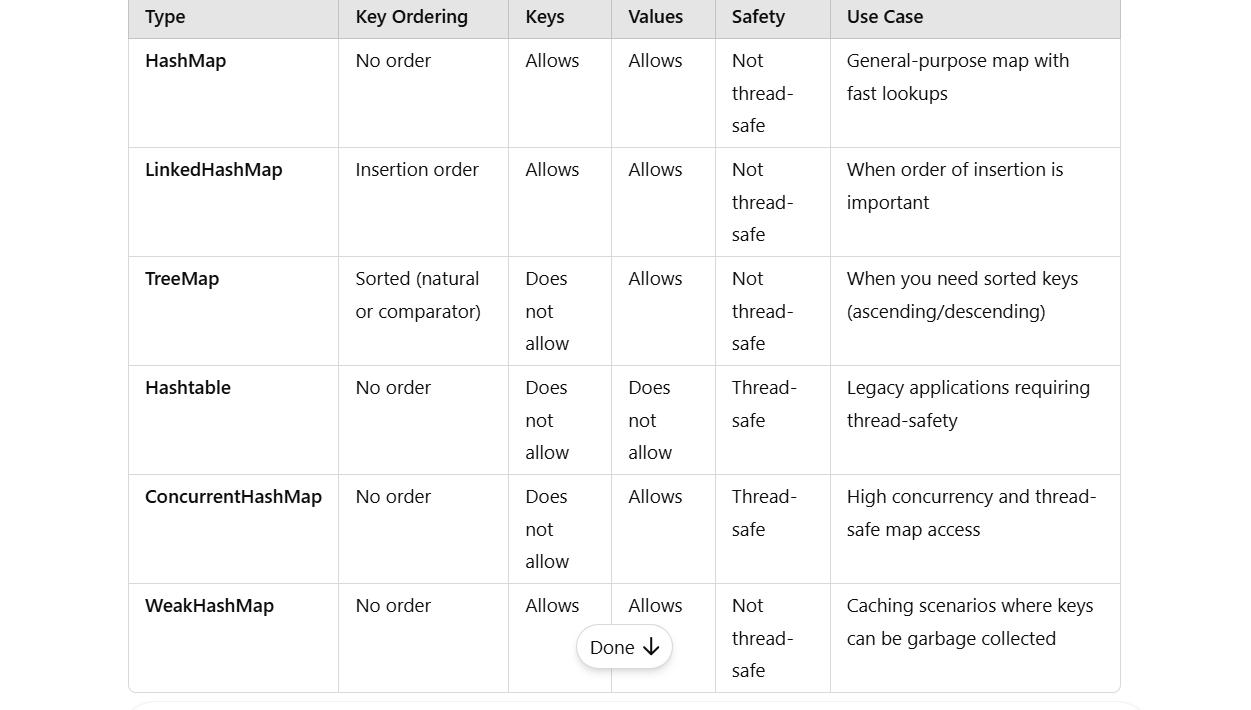
* **HashMap**: Does **not guarantee any order** of the keys or values. The entries can be in any random order.
* **TreeMap**: **Maintains a sorted order** of the keys, either by their natural ordering (if the keys are Comparable) or by a custom comparator provided at the time of map creation.

**4. Null Keys and Values**

* **HashMap**: Allows **one null key** and any number of null values.
* **TreeMap**: Does **not allow null keys** (since it requires comparison for ordering), but allows **null values**.

**6. Memory Overhead**

* **HashMap**: Has less memory overhead because it uses a hash table for storage.
* **TreeMap**: Has more memory overhead due to the tree structure.



How does HashMap work internally in Java?

**HashMap** uses a **hash table** for storage, it refers to the underlying mechanism for storing and retrieving key-value pairs efficiently.

A **hash table** is a data structure that stores data in an array-like format(An array of buckets), but it allows for fast data retrieval using a **hash function**. This Hash function converts the **key** into a **hash code** which determines the bucket index they key should store into.

**Hashing** is the process of converting a **key (like a string)** into a **fixed-size number** (called a *hash code*) using a function called a **hash function**.

**1.Hash Function**: A hash function takes the key and computes an integer (hash code) that represents that key. Hash function uses the hash code to determine the appropriate bucket index or slot where the keyvalue pair should be stored in the hash table.

Note: The bucket index (or bucket number) is usually derived by applying some additional operation (e.g., modulo) on the hash code.

**2.Collisions**: If two keys end up with the same hash code (i.e., they hash to the same index), this is called a **collision**. Different strategies are used to handle collisions internally by default:

* **Chaining**: The keys that hash to the same index are stored in a linked list or another data structure at that index.
* **Open Addressing**: If a collision occurs, the hash table looks for the next available slot in the array to store the key-value pair.

**3.Retrieval**: To retrieve the value, the hash function is applied to the key again, which gives the index where the value is stored. If there’s a collision, the hash table checks the linked list or probes the array for the correct key.

**Example:**

If we have a key, say "apple", the hash function might compute a hash code like 123456. If the hash table has 10 buckets, the bucket index would be determined by:

*bucketIndex* = 123456 % 10 = **6**

So, the key "apple" would be placed in bucket number **6**.

When two **different keys** (like "apple" and "orange") generate the **same hash code** in a hash function, this is called a **hash collision**. While the keys are different, the hash function may still generate the same numerical hash code for them. This can happen due to the limited range of possible hash values and the way hash codes are calculated.

**Why does this happen?**

1. **Finite Range of Hash Codes**:
   * Hash functions usually generate a finite range of hash codes (e.g., 32-bit or 64-bit integers). If there are more possible keys than the range of hash codes, multiple keys can end up producing the same hash code. This is an inherent property of hash functions, known as the **pigeonhole principle**: if you have more "pigeons" (**keys**) than "pigeonholes" (possible hash codes), some pigeons must share a pigeonhole (collide).

Let’s say we have two strings: "apple" and "orange".

* The hash function computes a hash code for each key (e.g., a 32-bit integer).
* After applying the hash function, both "apple" and "orange" might end up with the same hash code value, say 1234567 (even though the keys themselves are different).
* The hash table uses this hash code to determine the index of the bucket. Since both keys have the same hash code, they would initially map to the same bucket index.

When a **collision** happens (i.e., two different keys produce the same hash code), the hash table uses a technique to handle these collisions.

**Collision resolution** is **built-in** and **automatic**.

Two common methods for handling collisions are:

 **Chaining**:

* Each bucket holds one **list** (or another data structure, like a linked list) of key-value pairs. When a collision occurs, both "apple" and "orange" will be stored in the same bucket but as separate entries in the list.

On retrieval (get(key)), Java will:

* Go to that index,
* Traverse the linked list,
* Compare keys using equals() until it finds the right one.
* And gets the value for the key.

 **Open Addressing**:

* If a collision occurs, the hash table checks other nearby buckets for an available spot (this could involve probing, like linear probing or quadratic probing). If the bucket index for "apple" is already taken by "orange", the table will check other indices until it finds an empty one.

Does these all happen internally???

Yes, **all of these processes happen internally** within the **hash table** (or **hash map**) implementation. As a user or programmer interacting with the hash table, you typically don't need to worry about the specific internal details. The hash table automatically handles these operations for you, abstracting the complexity.

Here’s a breakdown of what happens internally in the case of a collision or inserting a key into the hash table:

**1. Hash Code Generation:**

* When you insert a key (like "apple" or "orange") into a hash table, the **hash function** is applied to that key. This is typically done automatically when you call the put or insert method.
* The hash function generates a **hash code** based on the key. This is an integer that represents the key in a compressed form.
* Internally, this might involve mathematical operations on the key (like converting it to an integer or performing bit shifts).

**2. Bucket Index Calculation:**

* The hash code generated by the hash function is then **mapped to a bucket**. The hash table determines the appropriate **bucket index** by taking the hash code and using modulo division (i.e., hashCode % numberOfBuckets).
* This mapping is done internally in the data structure, so as a user, you don't see or handle this calculation directly.

**3. Handling Hash Collisions:**

If two different keys generate the **same hash code**, the hash table needs to **resolve the collision**.

* **In Chaining** (common method), the hash table stores multiple entries in the same bucket, typically using a linked list or another data structure.
  + When a new key-value pair is inserted, if the bucket already contains an entry (i.e., a collision occurs), the hash table will add the new entry to the linked list or similar structure at that bucket.
  + This process is **transparent to the user**—you just get the behavior of the key-value pair being inserted or retrieved.
* **In Open Addressing** (another common method), if a collision occurs, the table looks for an **open slot** in other nearby buckets. The table will search the hash table based on a probing strategy (linear, quadratic, or double hashing), trying to find an empty slot for the key-value pair.
  + This process also happens internally, and you don’t need to manually manage it.

**4. Retrieving Keys:**

* When you try to retrieve a value using a key, the hash table follows a similar process:
  1. It computes the hash code for the key.
  2. It calculates the **bucket index** using the same hash code.
  3. It checks the bucket to see if the key exists:
     + If there are multiple entries in the bucket (due to chaining), the table will search through the list for the matching key.
     + If open addressing is used, it will search through nearby buckets for the key.

**5. Resizing (if necessary):**

* When the hash table grows too full (i.e., it exceeds a certain load factor), it automatically **resizes** itself internally. This typically involves creating a larger array for the buckets and **rehashing** all existing keys into the new array to ensure efficient access.

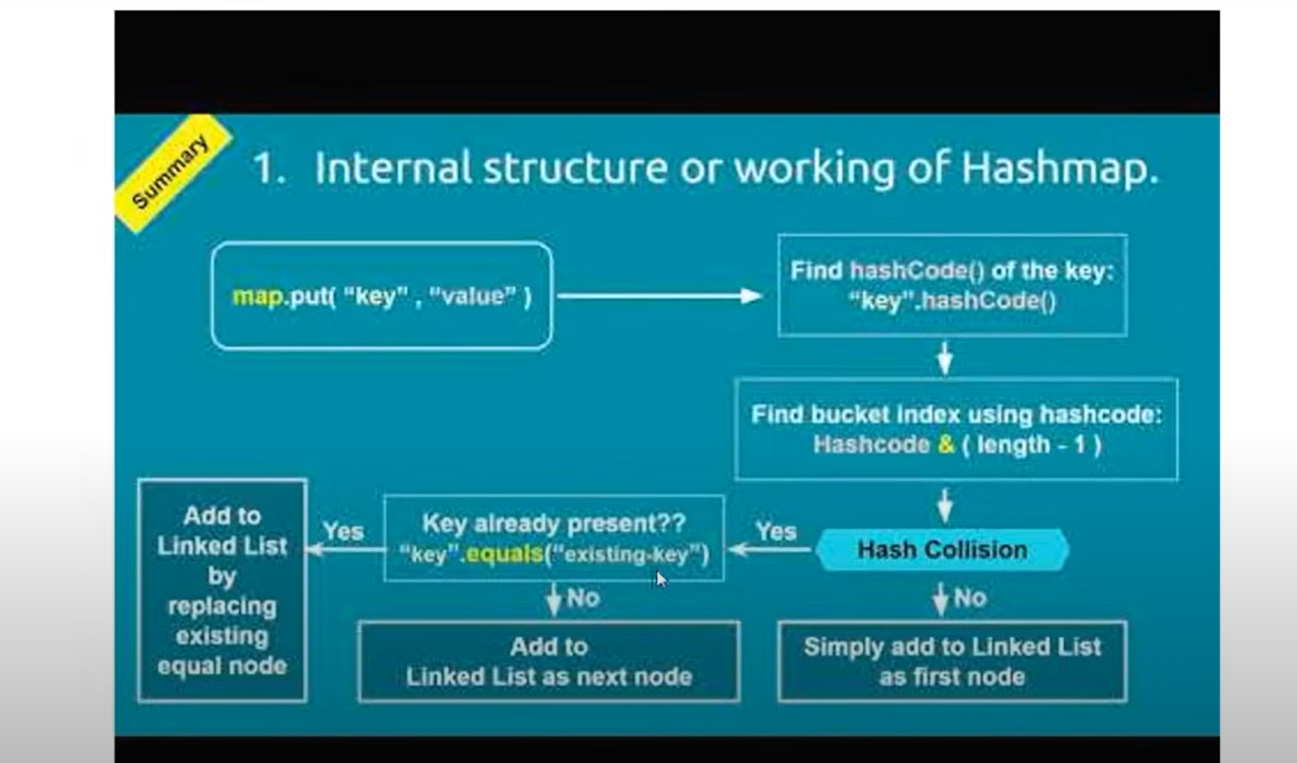
**Conclusion:**

Yes, **all of these processes happen internally** in the hash table or hash map. The key operations—hash code generation, bucket index calculation, collision resolution, and rehashing—are handled automatically by the data structure, so you can focus on simply interacting with the hash table (e.g., inserting and retrieving values).

**Final Summary (Corrected)**

A screenshot of a computer

AI-generated content may be incorrect.



When a HashMap is created, it has 16 empty buckets.  
When you put(key, value), Java computes a **hash** from the key’s hashCode(), spreads the hash, and calculates a **bucket index**.  
The entry is stored in that bucket.  
If another key ends up in the same bucket (due to hash collision), Java uses a **linked list or tree** to store multiple entries.  
Java handles collisions internally and maintains efficient access using this structure.

In a HashMap in Java, if the key is null, the key-value pair is **always stored in bucket 0 (i.e., index 0 of the internal array)**.

**MultiThreading:**

**Multithreading** means that your program can **run multiple tasks at once** by creating multiple threads.

Threads allow a program to do more things **at once**, and it makes your program faster and more responsive.

This way, the program doesn't have to wait for one task to finish before starting the next Instead, they **switch between each other quickly** (in **concurrency**) or **run at the same time** (in **parallelism**).

**Example:**

* Imagine you’re **cooking dinner** and **doing laundry** at the same time.
* If you do both tasks **concurrently**, you don’t have to finish one before starting the other. You can switch between them: start the laundry, then stir the pot, then go back to the laundry, and so on.

**threads** are like the tasks you’re switching between. You can have multiple threads (tasks) running, and the program switches between them.

**How Do Threads Run?**

* In a **single-core CPU** (like many older computers), **only one thread can run at a time**. But the operating system switches between threads so fast that it seems like they’re running **simultaneously**.
  + **For example**: Thread 1 starts, then Thread 2 starts, then Thread 1 resumes, and so on. The CPU switches between them very quickly, giving the appearance of them running at the same time.
* On a **multi-core CPU** (like modern computers), the CPU can actually run **multiple threads at the same time**, using different cores for each thread.

**What’s the Difference Between Concurrency and Parallelism?**

**Concurrency**

* **Concurrency** is when you have multiple tasks, but they don’t run exactly at the same time. They **switch** between each other quickly by CPU.
* Eg: Imagine you’re cooking and doing laundry. You can't do both tasks at the exact same time, but you switch between them quickly (stir the pot, load the laundry, stir the pot, load the laundry). This is concurrency — the tasks are managed and switched between, but only one task is happening at any given moment.
* Happens on single-core processors (because there’s only one core, so the CPU switches between threads).

**Parallelism**

* **Parallelism** is when tasks **actually run at the same time** on different cores of the CPU.
* Imagine you have two workers: one cooking and one doing laundry. Now, they are both working **at the same time**, not just switching between tasks.

**In simple terms**:

* **Concurrency** = Switching between tasks quickly (happens on single-core CPU).
* **Parallelism** = Tasks run at the same time (happens on multi-core CPU).

Multithreading in Java is a programming concept that allows multiple threads (or small units of a process) to run concurrently. Each thread represents an independent path of execution within a program. By using multithreading, Java programs can perform multiple tasks simultaneously, which can improve performance, especially for tasks that are independent of one another.

In multithreaded applications, **threads can interfere with each other**, leading to data inconsistency. To prevent this, Java provides mechanisms for **synchronization**, which allows only one thread to access a particular resource at a time.

** Single Thread:** One task runs at a time**.**

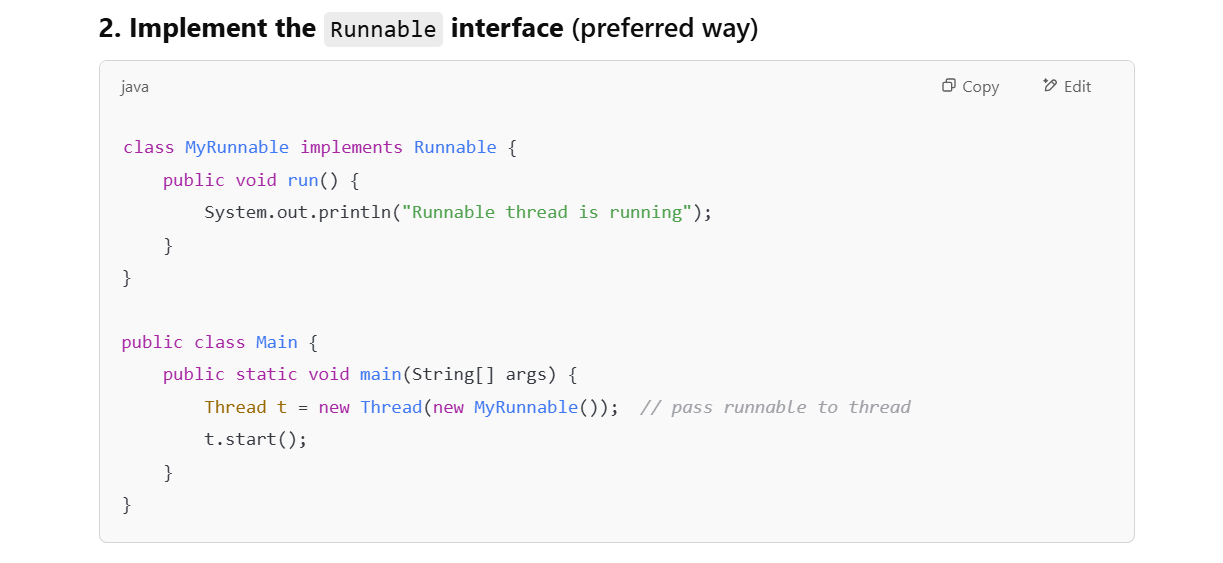
** Multiple Threads:** Multiple tasks run at the same time or concurrently.

 **Synchronization** is about controlling access to shared resources between threads to avoid conflicts or unexpected behavior (like race conditions)(i,e) to ensure threads don’t interfere with each other.

 **Synchronization does not prevent concurrency**. It just limits the **concurrency** for the **synchronized** code, but **other parts** of the program can still run concurrently without synchronization.

🔨 3 **Main Ways to Create a Thread in Java**





A screenshot of a computer program

AI-generated content may be incorrect.

**🚫 Don't Use .run() Directly**

Always use .start() to start a thread, not .run().

Calling .run() just runs the method in the **same thread** (like a normal method call), while .start() creates a **new thread**.

**A white box with black text

AI-generated content may be incorrect.**

**💤 Thread.sleep() in Java**

**🔹 What it does:**

Thread.sleep(milliseconds);

This **pauses the current thread** for a specified number of milliseconds.

**✅ Key points:**

* The **thread goes to sleep**, meaning it temporarily **stops executing**.
* It lets **other threads run** during that time.
* After the time is up, the thread is **resumed automatically**.
* It **does NOT stop the whole program**, only the **current thread**.

Adding **sleep()** in threads helps you **see them interleave** more clearly — otherwise, threads may finish too fast to observe concurrency.

**⚠️ Important:**

* Thread.sleep() can throw a **checked exception**: InterruptedException
  + That’s why we use a try-catch block around it.

**Multithreading in Java Collections**

Java Collections, by themselves, are not inherently thread-safe.

**Thread-Safe Collections**

* **Concurrent Collections:** Java provides a set of collections that are designed for use in multithreaded environments. These collections are part of the java.util.concurrent package. Some of the commonly used concurrent collections are:
  + **ConcurrentHashMap**: A thread-safe implementation of a map that allows concurrent read and write operations.
  + **CopyOnWriteArrayList**: A thread-safe variant of ArrayList where changes (such as add, remove) result in a new copy of the underlying array.
  + **BlockingQueue**: A queue that supports operations that block the thread until the operation can proceed, like put() and take().
  + **ConcurrentLinkedQueue**: A thread-safe, non-blocking queue.

These collections handle synchronization internally and do not require external synchronization to ensure thread safety.

**How a Program is Considered Single-Threaded or Multi-Threaded:**

The **threaded nature** of the program depends on how you structure it:

* **Single-Threaded Program**: The default Java program is single-threaded unless you explicitly create new threads.
* Only one thread runs the program that is main(). All operations are done sequentially one by one by this single thread.
* They can only perform one task at a time.
  + This is usually the case for simple programs where the execution flow happens step by step in one thread.
* **Multi-Threaded Program**: If you create new threads in your program, it becomes multi-threaded.
  + This can be done using the Thread class, ExecutorService, Runnable or other concurrency utilities.

**Why Use Multi-threading If a Single Thread Can Do All the Work?**

At first glance, it might seem like **single-threaded** programs should be enough because one thread can do all the work sequentially. However, there are some **important reasons** why multi-threading is used in programs, especially when **performance** or **responsiveness** is a concern.

**Key Reasons to Use Multi-threading:**

1. **Performance Improvement (Parallelism)**:
   * **Single-threaded programs** can only perform one task at a time. Even if your computer has **multiple CPU cores**, a single-threaded program will only use one core, meaning that it doesn't take full advantage of the hardware.
   * **Multi-threading** allows you to break a task into smaller parts, with each part running in its own **thread**. If your machine has multiple **cores**, you can run different threads on different cores, which can **speed up** the program. This is called **parallelism**.
   * For example, imagine you're sorting a massive list of numbers. If you break the list into smaller chunks and process them in **parallel threads**, it can finish faster than processing them one by one.
2. **Responsiveness (Non-blocking operations)**:
   * In a **single-threaded program**, if one task is slow or takes time (like downloading a large file or waiting for user input), it can **block** the whole program. This means everything else in the program **waits** for that task to finish.
   * In **multi-threaded programs**, different tasks can run in parallel. For example, you can have one thread downloading a file while another thread handles user input. This **keeps the program responsive**, and users don’t have to wait for the file download to complete before interacting with the program.
   * This is especially important in **real-time applications** like **games**, **web servers**, or **UI applications**, where users expect immediate feedback.
3. **Background Tasks**:
   * Some tasks don't need to run immediately but still need to be done. For example, a **background task** might be downloading updates or processing data while the main program continues to run and interact with the user.
   * With **multi-threading**, you can perform these tasks **in the background** without freezing or slowing down the main thread of the program.
4. **Better Resource Utilization**:
   * **Multi-threading** helps utilize **multiple cores** of the processor effectively. Most modern processors have **multiple cores** (like **quad-core**, **octa-core**, etc.). By using multiple threads, your program can run on different cores simultaneously, which improves performance.
   * If you're running a program on a **multi-core CPU**, a multi-threaded program can run faster because each thread can be handled by a different core, **reducing the overall execution time**.
5. **Improved Throughput for I/O-bound Tasks**:
   * For programs that involve a lot of **I/O operations** (e.g., reading and writing to a file, accessing a database, making network requests), multi-threading can be used to handle multiple I/O tasks **concurrently**.
   * I/O operations tend to be **slow** compared to processing tasks, and in a **single-threaded program**, the thread would have to wait for each I/O operation to complete before moving on to the next task.
   * In a multi-threaded program, while one thread waits for an I/O operation to finish, other threads can continue performing useful work.

**Real-Life Example:**

Let’s take the example of a **Web Browser**.

* In a **single-threaded browser**, everything would have to wait for each operation to finish before moving to the next. For example:
  + You click on a link, and the browser has to **load the page**. During that time, you can't scroll the page, click on anything else, or interact with the browser because it's all happening in one thread.
* In a **multi-threaded browser**, when you click on a link to load a page, the browser can:
  + Continue displaying and interacting with the current page in the main thread.
  + Load the new page in a **background thread** without blocking the user’s interaction.
  + Even load other resources (images, scripts, etc.) in separate threads, allowing you to use the browser smoothly while everything is happening in parallel.

**Types of Programs Where Multi-threading Is Useful:**

1. **Web Servers**: A web server needs to handle many users simultaneously. Each user’s request can be processed in a separate thread.
   * Without multi-threading, the server would have to process one request at a time, leading to **slower responses** for users.
2. **Games**: Games need to handle different things simultaneously—user input, physics, rendering, sound, and AI. Multi-threading allows these tasks to run simultaneously, improving game performance and responsiveness.
3. **User Interfaces (UI)**: In GUI programs, you don’t want the UI to freeze while a task is running (like file loading or complex calculations). By offloading long-running tasks to background threads, the UI remains **responsive**.
4. **File and Network I/O**: Programs that do a lot of reading/writing (e.g., downloading files, processing big data, or accessing databases) can use multi-threading to perform multiple I/O tasks concurrently without blocking the rest of the program.

**Single-Threaded vs Multi-Threaded Example:**

**Single-Threaded:**

Imagine a **file downloader** in a single-threaded program:

* The program downloads the file from the internet.
* While downloading, you **cannot** do anything else in the program, like interact with the UI or click buttons.

**Multi-Threaded:**

In a **multi-threaded** version of the same program:

* One thread handles the downloading of the file in the background.
* The main thread continues to **update the UI**, allowing you to cancel the download, show progress, or perform other tasks while the file is still downloading.

**Summary:**

While a **single-threaded program** works perfectly for small, simple tasks or applications, **multi-threading** offers **significant benefits** for performance, responsiveness, and resource utilization. It allows programs to **handle multiple tasks simultaneously**, especially when dealing with I/O operations, complex processing, or user interactions that need to happen concurrently.

* **Single-threaded**: Simple, straightforward, one task at a time.
* **Multi-threaded**: Efficient, handles multiple tasks concurrently, improves performance and user experience.

If your program doesn't need to handle multiple tasks at once, single-threading might be fine. But for **real-world applications** like web servers, games, and interactive UIs, **multi-threading** is essential to handle complex operations efficiently.

**When is Multi-threading Useful?**

* **Time-consuming tasks**: When a task involves waiting (like downloading data, reading from a file, or network requests), you can perform other tasks while waiting.
* **Improving performance**: By using multiple threads, you can perform tasks in parallel, especially if you have a multi-core processor.
* **Concurrency** allows us to run two tasks at the same time without having to wait for one to finish before starting the other.

When multiple threads access or modify data at the same time, thread-safety becomes an issue. Without proper synchronization or protection, threads can interfere with each other, causing errors or incorrect results. This is where **ConcurrentHashMap** and other thread-safe data structures come in—they allow multiple threads to work with the data safely without causing corruption or inconsistency.

**Can we use HashMap in a multi-threaded environment?**

Technically, you **can** use HashMap in a multi-threaded environment, but **it's not safe** for concurrent use in most cases. If multiple threads are accessing or modifying a HashMap simultaneously without proper synchronization, it can lead to **data corruption**, **inconsistent behavior**, or **exceptions**.

Here’s why:

**1. Lack of Thread Safety:**

* HashMap is **not synchronized**, meaning if one thread is modifying the map (e.g., put(), remove(), replace()) while another thread is reading or modifying the map, it can result in **race conditions**. This could cause inconsistent or incorrect results.
* For example, if two threads try to add an entry to the map at the same time, one of the threads might overwrite the other’s data, or the map might end up in an inconsistent state.

**2. Concurrent Modifications:**

* If multiple threads are iterating over the map while it’s being modified by another thread, it can cause issues like **ConcurrentModificationException**. The iterator of the HashMap doesn't handle changes made by other threads while it’s iterating.

**Example of Possible Issues:**

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

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

// Thread 1

Thread thread1 = new Thread(() -> {

map.put("A", 1);

map.put("B", 2);

});

// Thread 2

Thread thread2 = new Thread(() -> {

map.put("C", 3);

System.out.println(map.get("A")); // May cause issues if thread1 modifies map while reading.

});

thread1.start();

thread2.start();

}

}

**Problems with this code:**

* There’s no synchronization between the threads. If thread1 modifies the map while thread2 reads it (or modifies it), it can lead to unexpected results or even runtime exceptions.

**How to Safely Use HashMap in a Multi-threaded Environment:**

If you still want to use HashMap in a multi-threaded environment, you **must** ensure that access to the map is synchronized. Here are some approaches:

**Synchronization**:

Synchronization is a mechanism to control access to a shared resource by multiple threads. It ensures that **only one thread** can access a resource at a time, preventing **race conditions** and data inconsistency.

A synchronized map means access to the map (read/write) is wrapped with synchronized blocks to prevent concurrent modification by multiple threads.

Internally, every method call (e.g., put, get, etc.) is synchronized, which ensures **thread safety**, but **at the cost of performance** because only one thread can operate on the map at any given time.

**🧠 Think of it Like a Bathroom:**

* The house (program) has **many people** (threads) living in it.
* They can all do different things **at the same time**: eat, sleep, code.
* But there's only **one bathroom** (critical section), and it's **locked** when in use.
* So only **one person (thread)** can use it at a time — **that’s synchronization**.
* Still, **other people (threads)** are free to do **other things**.

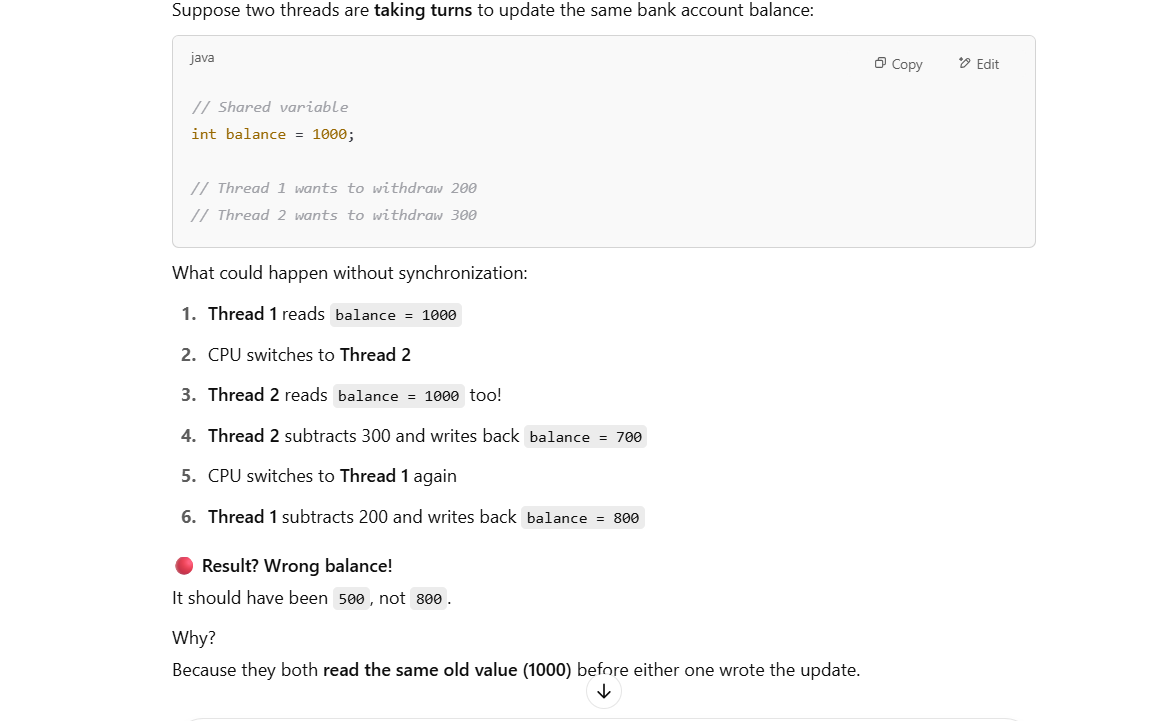
**🤔 If Threads Take Turns, Why Do We Need Synchronization?**

Yes, in a **single-core CPU**, threads take turns. But here’s the **key point**:

Even though they take turns **executing**, they can **interrupt each other at *any point***—including in the **middle** of reading or writing a shared resource (like a variable, file, or database).

**🧠 Think of It Like This:**

Suppose two threads are **taking turns** to update the same bank account balance:



 Threads can **read the same data**, but then each one performs operations without knowledge of the other’s actions.

 **Thread 2** doesn’t automatically see the updated value because **Thread 1**'s update isn’t visible yet — the two threads are essentially "operating in their own little world" and are unaware of each other’s changes until synchronization or coordination happens.

When we **synchronize** access to the shared resource, we ensure that **only one thread** can perform its read-modify-write cycle at a time.

Now, **Thread 1** will **lock** the balance for its operations. If **Thread 2** tries to access it while **Thread 1** is inside the synchronized block, **Thread 2** will have to wait until **Thread 1** finishes.

Thus, **Thread 2** will always see the updated value and won't overwrite the changes made by **Thread 1**.

**1. Synchronize the Map Explicitly:**

* You can synchronize access to the entire map using synchronized blocks. This ensures that only one thread can modify or access the map at a time, which prevents concurrency issues.

import java.util.HashMap;

public class SynchronizedHashMapExample {

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

public static void main(String[] args) {

// Synchronize on the map object for thread-safe access

Thread thread1 = new Thread(() -> {

synchronized (map) {

map.put("A", 1);

map.put("B", 2);

}

});

Thread thread2 = new Thread(() -> {

synchronized (map) {

map.put("C", 3);

System.out.println(map.get("A"));

}

});

thread1.start();

thread2.start();

}

}

* **Disadvantage**: The entire map access is locked, meaning only one thread can operate on the map at a time. This can cause performance issues if the map is heavily accessed by multiple threads.

**2. Using Collections.synchronizedMap():**

* You can wrap the HashMap with Collections.synchronizedMap() to make the map thread-safe. This creates a synchronized version of the HashMap, where every method call is synchronized.

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class SynchronizedMapExample {

public static void main(String[] args) {

Map<String, Integer> map = Collections.synchronizedMap(new HashMap<>());

// Thread 1

Thread thread1 = new Thread(() -> {

map.put("A", 1);

map.put("B", 2);

});

// Thread 2

Thread thread2 = new Thread(() -> {

map.put("C", 3);

System.out.println(map.get("A"));

});

thread1.start();

thread2.start();

}

}

* **Disadvantage**: While this makes the map thread-safe, it still synchronizes the entire method call, potentially creating bottlenecks and limiting concurrency.

**3. Using ConcurrentHashMap:**

* If you need to support high concurrency, the best option is to use **ConcurrentHashMap**. It is designed for high-concurrency use cases, allowing multiple threads to read and modify the map concurrently with minimal contention.

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {

public static void main(String[] args) {

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

// Thread 1

Thread thread1 = new Thread(() -> {

map.put("A", 1);

map.put("B", 2);

});

// Thread 2

Thread thread2 = new Thread(() -> {

map.put("C", 3);

System.out.println(map.get("A"));

});

thread1.start();

thread2.start();

}

}

* **Advantage**: ConcurrentHashMap provides **thread-safe** operations without the performance bottleneck of locking the entire map. It allows concurrent reads and writes with fine-grained locking, improving scalability and performance.

**When to Use ConcurrentHashMap Instead of HashMap:**

* **High-concurrency environments**: Where multiple threads will frequently read from and modify the map simultaneously.
* **Avoiding external synchronization**: ConcurrentHashMap handles internal synchronization efficiently, so you don't need to manually synchronize blocks or methods.

**Summary:**

* **HashMap is not thread-safe** for use in a multi-threaded environment. If multiple threads are modifying or accessing the map concurrently, it can lead to data inconsistency, race conditions, or runtime exceptions.
* If you need to use HashMap in a multi-threaded environment, you must either **synchronize** access using external synchronization mechanisms (like **synchronizedMap**() or **synchronized** blocks) or use a more appropriate data structure like **ConcurrentHashMap**.
* For **high-concurrency scenarios**, ConcurrentHashMap is usually the best choice.
* For multithreaded environments, prefer **ConcurrentHashMap** over **HashMap + synchronization.**

**Concurrent Hash Map:**

**ConcurrentHashMap** is a thread-safe variant of the HashMap that allows concurrent access to the map without requiring external synchronization.

It is part of the **java.util.concurrent** package and was introduced in Java 5.

**ConcurrentHashMap** helps in **allowing multiple threads** to **perform the same operation (eg: put) concurrently** on **different parts** of the map, without blocking each other. However, its ability to allow multiple threads to access the map and perform operations concurrently is tied to how the map is **internally structured** and how it handles **concurrency**.

**Key Features:**

1. **Thread-Safe**: It allows multiple threads to read and write to the map simultaneously without causing data inconsistency or data corruption, making it ideal for use in concurrent applications.

Unlike a regular HashMap, a ConcurrentHashMap ensures that **concurrent** **operations** (such as put, get, remove, etc.) do not corrupt the underlying data.

1. **Segmented Locking**: Unlike the standard HashMap, which locks the entire map during operations, In earlier versions of Java (before Java 8), ConcurrentHashMap uses a technique called **segmented locking** (or fine-grained locking). The map is divided into several segments, and each segment can be locked independently, allowing multiple threads to access different segments concurrently.( Each segment had its own lock, which allowed threads to lock only the segment they were operating on, **not the entire map**.)This **fine-grained locking** meant that multiple threads could safely update different parts of the map at the same time without blocking each other. However, if two threads tried to access the same segment, they would be **blocked** by the segment lock. The map was divided into 16 segments by default. So, if you had 16 threads, each thread could be working on a different segment without blocking others.

**Example**:

 If Thread 1 and Thread 2 were both trying to update different keys in the map (which were located in **different segments**), they could do so without blocking each other.

 If Thread 1 and Thread 2 tried to update keys in the **same segment**, they would be blocked, as only one thread could access the segment lock at a time.

**Bucket-Level Locking / Lock Striping**:

In modern versions of Java (Java 8 and later), The map was redesigned to eliminate the segment concept and instead use a **bucket-level locking** or **lock striping** mechanism which means different threads can concurrently read or write to **different buckets** of the map without blocking each other.

This means that instead of having one large lock per segment, ConcurrentHashMap now locks smaller, individual **buckets** (or entries) in the map.

ConcurrentHashMap uses **atomic operations** for common methods. For example, the methods used in your code (**putIfAbsent**, **replace**, etc.) ensure that updates are done in an atomic fashion, meaning they are executed in a way that guarantees consistency even when multiple threads are trying to perform the same operation concurrently.

1. **No Blocking Reads**: It allows multiple threads to perform **read operations** (get) without locking, even if other threads are modifying the map. However, write operations (like put or remove) are handled with locks to prevent conflicts.
2. **Atomic Operations**: Many operations on ConcurrentHashMap are atomic, such as putIfAbsent(), remove(), and replace(). These allow safe modifications of the map in a multi-threaded environment.
3. **Scalable**: It is designed to perform well in highly concurrent environments, with better scalability compared to using Collections.synchronizedMap() or manually synchronizing a regular HashMap.

**putIfAbsent(key, value)**: This method atomically checks if the key is absent and only adds the value if it's not already present. If multiple threads attempt to put a value for the same key, only one thread will succeed, and the others will see that the key is already present.

In a scenario like this:

map.putIfAbsent("Apple", 10);

This operation is **atomic** and thread-safe:

* If another thread has already inserted the value "Apple", then putIfAbsent will not modify the value.
* If no thread has added "Apple", it will be inserted without any race conditions, even if other threads are attempting to do the same.

This guarantees that no matter how many threads are trying to insert "Apple", only one thread will succeed in adding it, and others will simply see that it's already present.

In a **single-threaded program**, there is **no concurrency** because there is only one thread running, and tasks are executed one after another.

You typically use **ConcurrentHashMap** when you are working in a **multi-threaded** environment where multiple threads might be accessing and modifying the map concurrently, such as in a server handling multiple requests or in background tasks processing data.

**ConcurrentHashMap** helps in **allowing multiple threads** to **perform the same operation concurrently** on **different parts** of the map, without blocking each other. However, its ability to allow multiple threads to access the map and perform operations concurrently is tied to how the map is **internally structured** and how it handles **concurrency**.

**Key Concepts in ConcurrentHashMap and Multi-threaded Access:**

1. **Fine-Grained Locking (Segmented Locking):**
   * In older versions of ConcurrentHashMap (pre-Java 8), the map was divided into **segments**. Each segment had its own lock, allowing multiple threads to access and modify different segments concurrently. This was an early form of **fine-grained locking**.
   * In Java 8 and later, the internal structure has been improved to allow **bucket-level locking** or **lock-free techniques**, which means different threads can concurrently read or write to **different buckets** of the map without blocking each other.
2. **Concurrency Without Locking the Whole Map:**
   * One of the key advantages of ConcurrentHashMap is that it **does not lock the entire map** for every operation. If multiple threads are accessing different keys (or even the same key but with separate operations), **they do not block each other** unless they are trying to modify the same part of the map (e.g., the same key).
3. **Atomic Operations:**
   * Operations such as put, compute, merge, and replace are **atomic** for each key in ConcurrentHashMap, meaning that these operations can be safely used by multiple threads concurrently, ensuring thread safety without external synchronization.
   * For example, if multiple threads try to increment a value for the same key using the compute or merge methods, ConcurrentHashMap will handle the concurrency internally.

**Example of Multiple Threads Accessing the Map Concurrently on Different Keys:**

Let's consider an example where multiple threads are concurrently putting key-value pairs into the map.

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentAccessExample {

public static void main(String[] args) {

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

// Thread 1: Adding keys "key1", "key2"

Thread thread1 = new Thread(() -> {

map.put("key1", 1);

System.out.println("Thread 1 put key1 = 1");

map.put("key2", 2);

System.out.println("Thread 1 put key2 = 2");

});

// Thread 2: Adding keys "key3", "key4"

Thread thread2 = new Thread(() -> {

map.put("key3", 3);

System.out.println("Thread 2 put key3 = 3");

map.put("key4", 4);

System.out.println("Thread 2 put key4 = 4");

});

thread1.start();

thread2.start();

}

}

**Key Points:**

* **Fine-grained concurrency**: Thread 1 and Thread 2 are updating **different keys** (key1, key2, key3, key4). ConcurrentHashMap allows these updates to happen concurrently without locking the entire map, which would block the threads.
* **Non-blocking**: Threads accessing **different keys** don't block each other. Both threads can update their respective keys simultaneously.

**What Happens if Multiple Threads Modify the Same Key?**

If multiple threads try to modify the **same key** (for example, updating "key1"), ConcurrentHashMap will handle it in a thread-safe way. However, it will not block **other threads** that are accessing **different keys**.

For example:

Thread thread3 = new Thread(() -> {

map.put("key1", 10); // Thread 3 updates "key1"

System.out.println("Thread 3 put key1 = 10");

});

Thread thread4 = new Thread(() -> {

map.put("key1", 20); // Thread 4 also tries to update "key1"

System.out.println("Thread 4 put key1 = 20");

});

In this case, even though both threads are updating "key1", the map will handle these updates **atomically** (not concurrently for the same key). Only **one thread** will successfully update the value for "key1", but the internal mechanisms of ConcurrentHashMap (like **CAS (Compare-And-Swap)** operations) will ensure that these updates do not lead to inconsistent data.

**Conclusion: Does ConcurrentHashMap Help When Multiple Threads Perform the Same Operation?**

Yes, **ConcurrentHashMap** helps in the following ways:

1. **Concurrent Operations on Different Keys**: If multiple threads perform the same operation (like put, get, compute, etc.) on **different keys**, ConcurrentHashMap allows them to do so **without blocking** each other.
2. **Atomic Operations**: Operations like putIfAbsent, compute, merge, and replace can be used to **safely update values atomically**, even when multiple threads are trying to modify the same key concurrently.
3. **Thread-Safe Access**: ConcurrentHashMap ensures that you don't need external synchronization, making it ideal for **high-concurrency scenarios** where multiple threads need to perform operations on the map concurrently without compromising thread safety or performance.

In short, **ConcurrentHashMap** allows for safe concurrent access to the map, and it is especially useful when you have multiple threads performing operations on the map, **whether they are working on different keys or the same key**. The internal concurrency mechanisms ensure that the map handles operations efficiently and safely.

What is the purpose of the ConcurrentHashMap in java?

The **purpose** of ConcurrentHashMap in Java is to provide a thread-safe, highly scalable, and efficient alternative to the regular HashMap when multiple threads are accessing and modifying the map concurrently.

The **ConcurrentHashMap** in Java is a part of the **java.util.concurrent** package and is designed to be a thread-safe alternative to HashMap. Its purpose is to allow multiple threads to read and modify the map concurrently without causing data corruption or inconsistency, making it ideal for situations where many threads need to access and update a shared map concurrently.

**Key Features:**

1. **Thread-Safety**: Unlike HashMap, which is not thread-safe and requires external synchronization for concurrent access, ConcurrentHashMap handles synchronization internally. This makes it much more efficient in a multi-threaded environment because it allows multiple threads to access different parts of the map concurrently.
2. **Locking Mechanism**: ConcurrentHashMap achieves concurrency by dividing the map into segments (buckets), and only locking the segment that is being modified. This means that multiple threads can work on different segments of the map simultaneously, which improves performance over locking the entire map.
3. **No Blocking on Reads**: Reads are generally not blocked, even if a write operation is happening on the map. This makes it highly suitable for scenarios where read-heavy operations are common.
4. **Atomic Operations**: It provides atomic methods such as putIfAbsent(), remove(), and replace(), which allow for thread-safe modifications without needing to manually synchronize the operations.
5. **Scalable Performance**: As more threads access different parts of the map, ConcurrentHashMap can provide better scalability compared to other synchronized collections like Hashtable.

**Typical Use Cases:**

* When you have multiple threads that need to access and update a shared map.
* In high-performance applications where minimizing contention is crucial.
* In situations where many threads read data but few write or modify the data.

In summary, ConcurrentHashMap is useful when you need a thread-safe map with high concurrency, offering efficient performance for both read and write operations in a multi-threaded environment.

**🧠 Detailed Explanation:**

**1. Thread Safety**

* **HashMap**: **Not thread-safe**. This means that if multiple threads modify a HashMap at the same time, it could result in **data corruption** or **unexpected behavior**. You would need to manually synchronize access (e.g., using Collections.synchronizedMap() or explicit locks).
* **ConcurrentHashMap**: **Thread-safe** by design. It allows **multiple threads** to read and write to the map concurrently without causing data corruption. It does this by using **fine-grained locking** (locking only specific segments of the map instead of the entire map) to allow maximum concurrency.

**2. Locking Mechanism**

* **HashMap**: There is **no internal locking** in HashMap. If you are working in a multi-threaded environment and need thread-safety, you would either need to:
  + Wrap it with Collections.synchronizedMap(), or
  + Manually synchronize code blocks.

However, this is not efficient, as synchronizing the entire map can slow down performance significantly.

* **ConcurrentHashMap**: Uses a **fine-grained locking mechanism**:
  + **Segment locking**: The map is divided into segments, and each segment has its own lock. This allows multiple threads to operate on different segments concurrently.
  + When threads access the same segment, they are locked, but other threads can access other segments without blocking.
  + This **reduces contention** and improves concurrency, making ConcurrentHashMap ideal for high-concurrency scenarios.

**3. Null Keys and Values**

* **HashMap**: Allows one null key and multiple null values.

map.put(null, "some value"); // Allowed

map.put("some key", null); // Allowed

* **ConcurrentHashMap**: **Does not allow null keys or values**. If you try to insert a null key or value, it will throw a NullPointerException.

map.put(null, "some value"); // Throws NullPointerException

map.put("some key", null); // Throws NullPointerException

**4. Iterator Behavior**

* **HashMap**: **Fail-fast iterator**. If the map is modified (i.e., added or removed entries) while iterating over a list either iterator or for each loop, a ConcurrentModificationException is thrown.

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

map.put("newKey", "newValue"); // Throws ConcurrentModificationException

}

* **ConcurrentHashMap**: **Weakly consistent iterator**. This means that if the map is modified during iteration, the iterator will not throw an exception, but it might not reflect all the changes (i.e., it might "see" inconsistent data).

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

concurrentMap.put("newKey", "newValue"); // No ConcurrentModificationException

}

**5. Performance**

* **HashMap** is generally **faster** than ConcurrentHashMap in single-threaded scenarios because there is no overhead of synchronization.
* **ConcurrentHashMap** might be **slower** than HashMap in single-threaded scenarios due to internal synchronization, but it shines in multi-threaded applications by allowing higher concurrency and avoiding thread contention.

**🧑‍💻 When to Use Which?**

* **Use HashMap**:
  + When you're working in a **single-threaded** environment or if you're manually managing thread-safety (for example, using synchronized blocks).
  + When you need to store null values or a null key.
* **Use ConcurrentHashMap**:
  + When you're working in a **multi-threaded** environment and you need thread-safety without blocking other threads unnecessarily.
  + When your application requires high concurrency, and multiple threads need to access or modify the map simultaneously.
  + When you're dealing with **high traffic** systems or real-time applications like web servers or distributed systems.

**👨‍💻 Example Code for ConcurrentHashMap:**

import java.util.concurrent.\*;

public class ConcurrentHashMapExample {

public static void main(String[] args) {

// Create a ConcurrentHashMap

ConcurrentHashMap<String, String> concurrentMap = new ConcurrentHashMap<>();

// Put some entries

concurrentMap.put("apple", "red");

concurrentMap.put("banana", "yellow");

// Accessing an entry

System.out.println(concurrentMap.get("apple")); // Output: red

// Iterating through entries

for (String key : concurrentMap.keySet()) {

System.out.println(key + ": " + concurrentMap.get(key));

}

// No ConcurrentModificationException during iteration

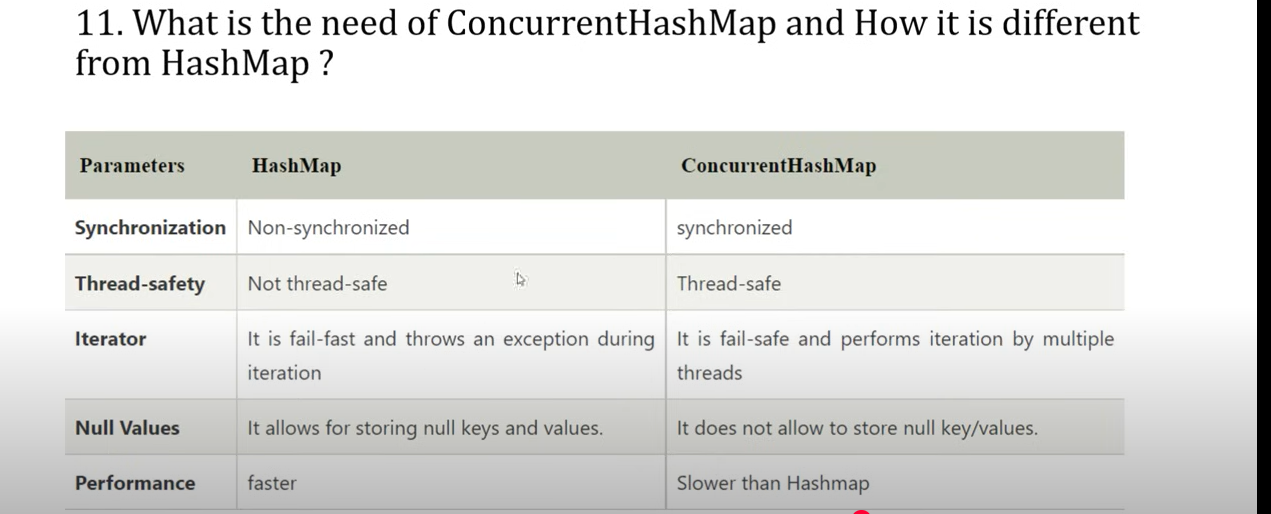
concurrentMap.put("cherry", "dark red");

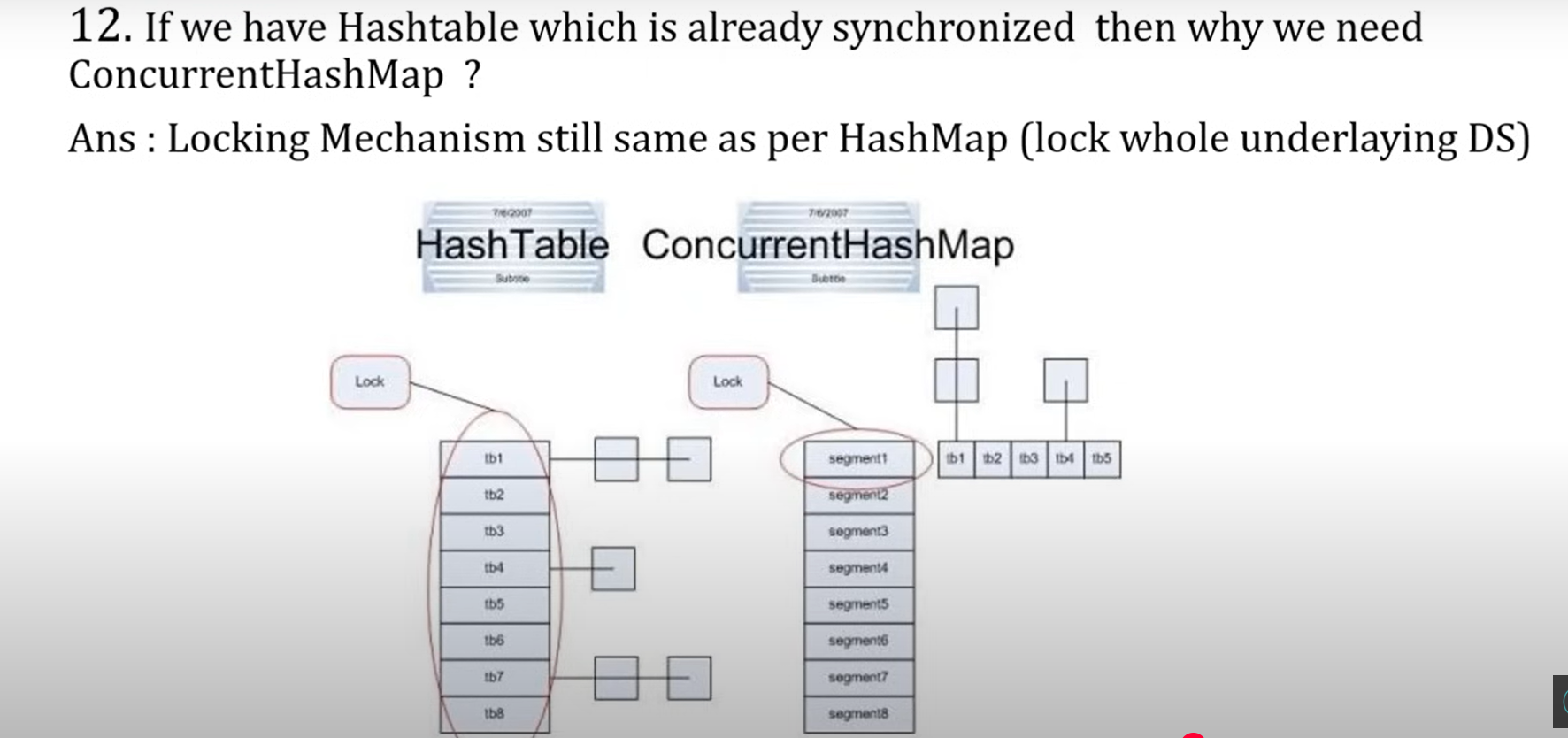
}

}

**🚀 Conclusion:**

* **HashMap**: Best for single-threaded environments or when you handle synchronization manually.
* **ConcurrentHashMap**: Best for multi-threaded environments where you need efficient and thread-safe concurrent access.





Hash table locks entire map whereas concurrent hash map does segment/bucket locking.

**Fail Fast Iterators:**

A **fail-fast iterator** is an iterator that immediately throws an exception (typically ConcurrentModificationException in Java) when it detects that the underlying collection has been modified (structurally modified) after the iterator was created (except through the iterator’s own methods like remove()).

**When it happens**: This typically occurs when you try to modify a collection (e.g., adding or removing elements) while iterating over it using an iterator.

The goal of a fail-fast iterator(throwing exception) is **to avoid potential inconsistencies** or undefined behavior that might occur if an iterator continues to work with a collection that has been modified while being iterated over.

**All the below are Fail-Fast Iterators:**

1. **Lists:**
   * **ArrayList**
   * **LinkedList**
2. **Sets:**
   * **HashSet**
   * **LinkedHashSet**
   * **TreeSet**
3. **Maps:**
   * **HashMap**
   * **LinkedHashMap**
   * **TreeMap**

**Fail fast:**

**Fail-fast,** the Iterator detects the structural modification (like removal of an element, add of an element ) and throws a **ConcurrentModificationException** to prevent potential inconsistent behavior during iteration.

Eg: 1

A screenshot of a computer program

AI-generated content may be incorrect.

If you want to avoid this exception while modifying the collection, you can use **Iterator.remove()** method to safely remove elements during iteration:

A screenshot of a computer program

AI-generated content may be incorrect.

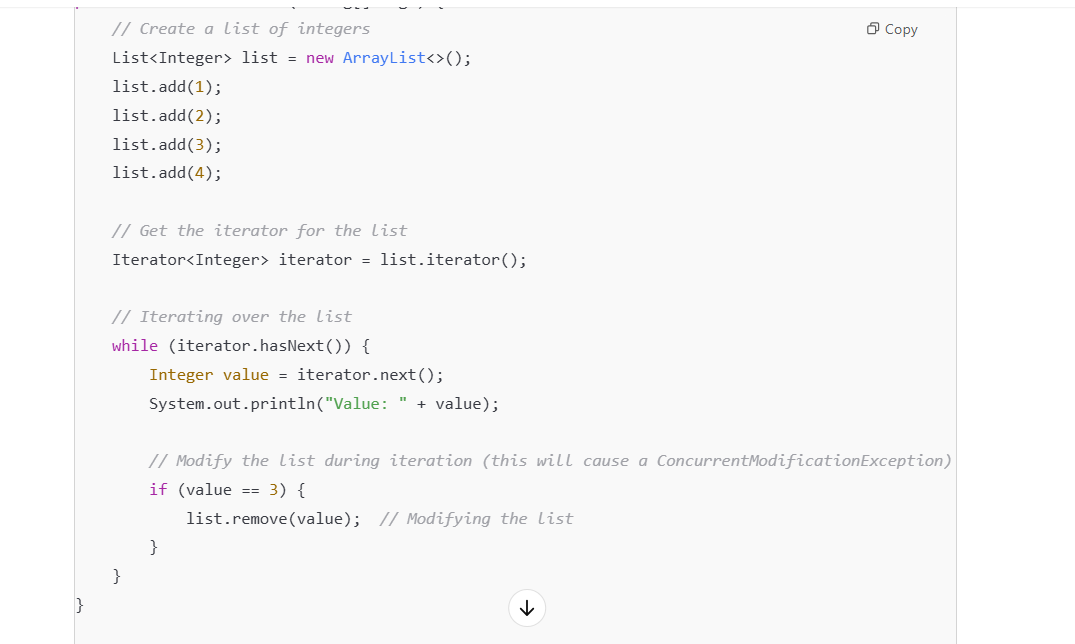
Eg:2

A screenshot of a computer program

AI-generated content may be incorrect.

A close-up of a computer screen

AI-generated content may be incorrect.



The **fail-fast behavior** is implemented by collections like ArrayList, HashMap, Vector, and others from the Java Collections Framework. When their underlying structure is modified outside the iterator (e.g., via add(), remove()), the iterator helps us to detect the modifications or unsafe operations and immediately throws a ConcurrentModificationException.

A screenshot of a computer

AI-generated content may be incorrect.

**🛠 How to Avoid It?**

* Use the iterator’s remove() method instead of modifying the collection directly.
* Use **Concurrent Collections** like CopyOnWriteArrayList, CopyOnWriteArraySet ConcurrentHashMap, etc., for thread-safe, fail-safe iteration.

**Fail-Safe Iterator:**

A **fail-safe iterator**, on the other hand, operates differently. It **does not throw a ConcurrentModificationException** even if the collection is modified during iteration. When an element is added, a new copy of the list is created. The internal reference to the list is updated to point to the **new copy**. However, **iterators that were created before the modification** still work with the original version (the snapshot from when they were created).

Eg: CopyOnWriteArrayList ,CopyOnWriteArraySet

CopyOnWriteArrayList is a thread-safe variant of ArrayList, designed to handle concurrency without locking. The key feature of CopyOnWriteArrayList is that it creates a copy of the underlying array when a modification (like add(), remove(), etc.) is made. This allows any modifications to the list to not affect the iteration process, ensuring safe traversal while the list is being modified by other threads.

 When you **create an iterator**, It simply **gets a reference** to the current internal array at that point in time. No new copy is made during reading — just a reference to the current one

The **copy of the list** is created **during a modification operation** (like add(), remove(), set()) — this is where the new array gets created, and **the reference to the live list is updated** to point to the new array.

but the iterator, which was created earlier, continues to reference the **old array** (the one it saw when it was first created).

 **The iterator works on the old snapshot** of the list (i.e., the original array before the modification). Therefore, it **does not see any changes** made to the list after it was created.

 The iterator internally **captures a reference to the current array** (the snapshot).

 Even if another thread modifies the list during the loop, **you still iterate over your own copy**, not the updated one.

**So What *Does* Happen to the Old Array?**

When CopyOnWriteArrayList updates its internal array reference (array = newArray):

The **old arrays remain in memory temporarily**, especially if another thread is still using them for reading or iteration.  
But once they’re no longer referenced, the JVM will garbage collect them.

**🧹 When Does the Old Array Get Garbage Collected?**

* Only after:
  + The iterator is no longer used
  + No other references to that snapshot exist
* Then the JVM will eventually GC the old array

So:

The **iterator "keeps the old list alive"** as long as it's needed.  
Once it's done, the old array is eligible for GC like any normal object.

Eg:1 – In single threaded program

A screenshot of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Eg:2 – In multithreading program

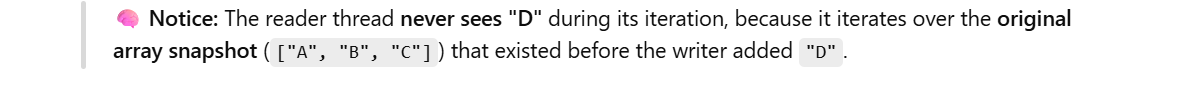


A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



✅ **In CopyOnWriteArrayList, any iteration (whether with for-each or Iterator) works on a snapshot** — the original array at the time the iteration started.

 If another thread **modifies the list** (adds/removes elements) *while you're iterating*,  
👉 you **won’t see those changes** during that iteration.

 You’re effectively reading from a **frozen copy** of the list — a snapshot.

In case of ConcurrentHashMap, it doesn’t create a copy of collection when collection is modified during iteration so it does print the element modified in the list.

**Why Use Fail-Safe Iterators?**

1. **Concurrent Environments**: Fail-safe iterators are particularly useful in **multi-threaded** environments, where one thread may be modifying the collection while another thread is iterating over it. A fail-safe iterator allows for modifications to occur without causing inconsistent behavior or exceptions in the iterating thread.
2. **Concurrent Collections**: Collections like CopyOnWriteArrayList, ConcurrentHashMap, and others in the java.util.concurrent package are designed to handle modifications during iteration by **working on internal copies**. These collections are designed for scenarios where you need to safely iterate over a collection while it may be concurrently modified by other threads.
3. **Avoiding Locks**: In multi-threaded applications, locking is often used to synchronize access to a collection, but fail-safe iterators allow iteration without needing to lock the collection. This can provide better performance in cases where reads are more frequent than writes.

**🛡️ Thread-Safety**

* All operations are **synchronized safely**.
* Iterators don’t need to be synchronized externally.
* It’s safe to iterate even while other threads are modifying the list — because iterators operate on a **snapshot** of the array.

No **ConcurrentModificationException** like you’d get with **ArrayList**.

**Summary:**

* **Fail-fast iterators** are best for scenarios where you want to catch unintended modifications to a collection during iteration, particularly in **single-threaded** environments or controlled multi-threaded environments.
* **Fail-safe iterators** are useful when you need to iterate over a collection that might be modified concurrently by multiple threads. However, they typically come with performance overhead, as they may work with copies of the collection.

**Common Fail-Safe Collections:**

* CopyOnWriteArrayList
* CopyOnWriteArraySet
* ConcurrentHashMap
* ConcurrentLinkedQueue

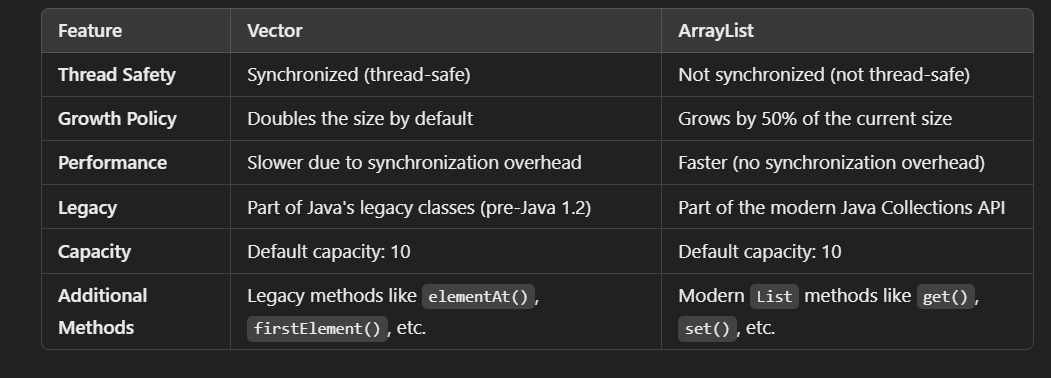
**Vector:**

Vector is a class that implements the List interface and is part of the Java Collections Framework. It is essentially a growable array of objects, similar to an ArrayList, but with some important differences, especially in terms of thread safety and the way it manages memory.

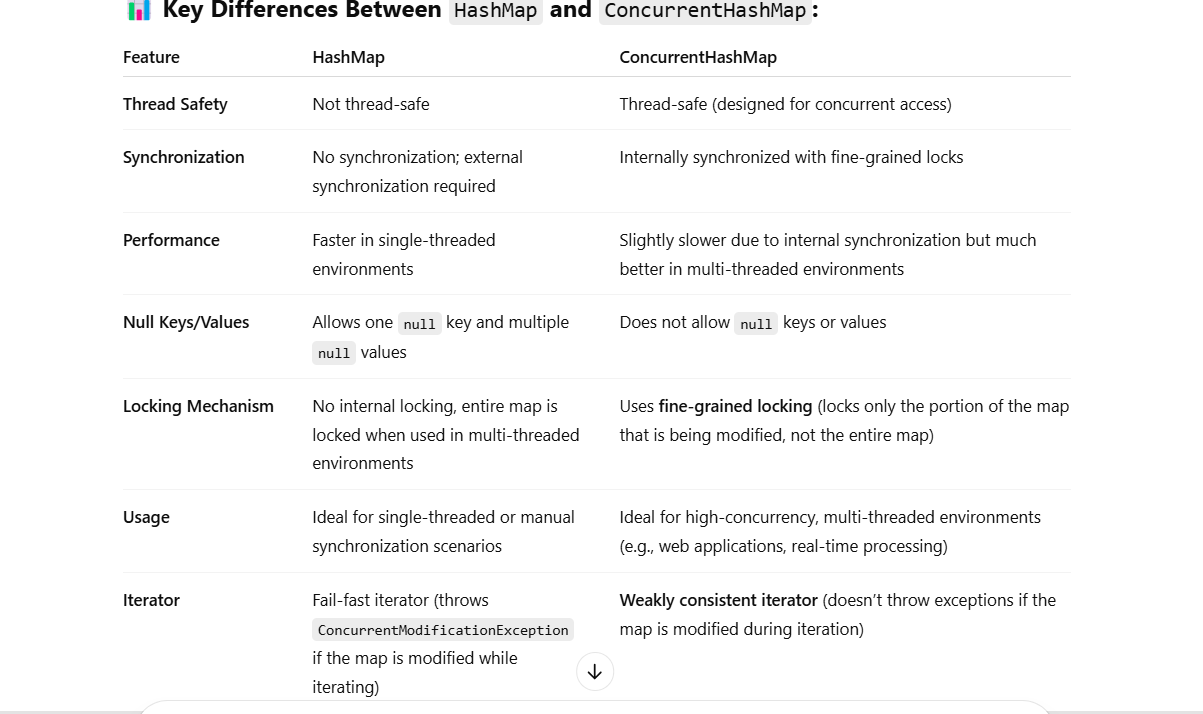
One of the key differences between a Vector and an ArrayList is that a Vector is **synchronized**, meaning that its methods are thread-safe by default. This allows multiple threads to access the Vector safely without additional synchronization. But this synchronization comes with a performance cost because only one thread can access the Vector at a time.

Vector doubles its size by default when it runs out of space, while ArrayList increases its size by 50% when it grows.

Vector is part of the **legacy** classes in Java, and while it is still supported, it is generally recommended to use ArrayList or other collections that are part of the **Java Collections Framework** for new projects because of vectors have slower performance due to synchronization overhead. ArrayList is faster but requires external synchronization if thread safety is needed.

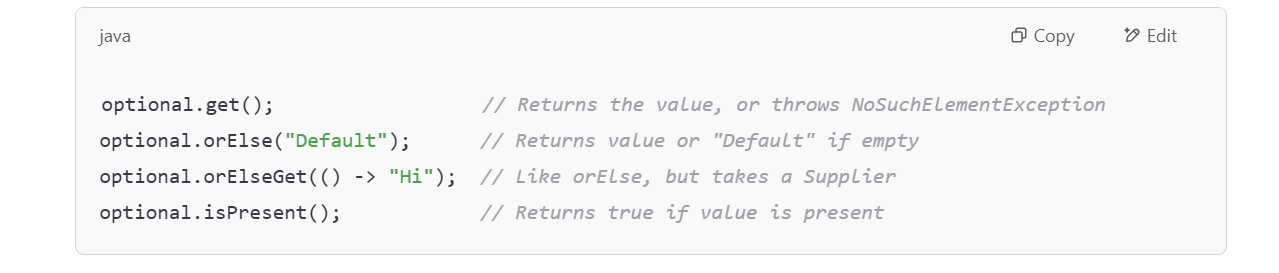


Differences between HashMap AND ConcurrentHashMap::



***Optional :***

In Java, Optional is a container object introduced in Java 8 that may or may not contain a non-null value. It's part of the java.util package and is commonly used to **avoid** **NullPointerException** and to represent optional or missing values more explicitly.

** **

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.