Collection Framework

```
class MyMap<K, V> {
    private static class Entry<K, V> {
        K key;
        V value;
        Entry<K, V> next;

        Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }

    private final int capacity = 16;
    private Entry<K, V>[] buckets = new Entry[capacity];

    private int hash(K key) {
        return (Objects.hashCode(key) & 0x7FFFFFFF) % capacity;
    }
```

```
public void put(K key, V value) {
    int index = hash(key);
    Entry<K, V> head = buckets[index];

    for (Entry<K, V> curr = head; curr != null; curr = curr.next) {
        if (Objects.equals(curr.key, key)) {
            curr.value = value;
            return;
        }
    }

    Entry<K, V> newEntry = new Entry<>(key, value);
    newEntry.next = head;
    buckets[index] = newEntry;
}

public V get(K key) {
    int index = hash(key);
    for (Entry<K, V> curr = buckets[index]; curr != null; curr = curr.next) {
        if (Objects.equals(curr.key, key)) {
            return curr.value;
        }
    }
    return null;
}
```

```
    public V remove(K key) {
        int index = hash(key);
        Entry<K, V> curr = buckets[index];
        Entry<K, V> prev = null;

        while (curr != null) {
            if (Objects.equals(curr.key, key)) {
                if (prev == null) {
                    // Removing head of the list
                    buckets[index] = curr.next;
                } else {
                    // Unlink the current node
                    prev.next = curr.next;
                }
                return curr.value;
            }
            prev = curr;
            curr = curr.next;
        }

        return null; // key not found
    }
}
```

It provides a set of interfaces and classes that help in managing groups of object.

Before the introduction of the Collection Framework in JDK 1.2, Java used to rely on a variety of classes like *Vector*, *Stack*, *Hashtable*, and *arrays* to store and manipulate groups of objects.

- *Inconsistency:* Each class had a different way of managing collections, leading to confusion and a steep learning curve.
- *Lack of inter-operability:* These classes were not designed to work together seamlessly.
- *No common interface:* There was no common interface for all these classes, which meant you couldn't write generic algorithms that could operate on different types of collections.

Using Collections –

- *Unified architecture:* A consistent set of interfaces for all collections.
- *Inter-operability:* Collections can be easily interchanged and manipulated in a uniform way.
- *Reusability:* Generic algorithms can be written that work with any collection.
- *Efficiency:* The framework provides efficient algorithms for basic operations like searching, sorting, and manipulation.

Key Interfaces in the Collection Framework

The Collection Framework is primarily built around a set of *interfaces*. Important ones are:

Collection: **The root interface for all the other collection types.**

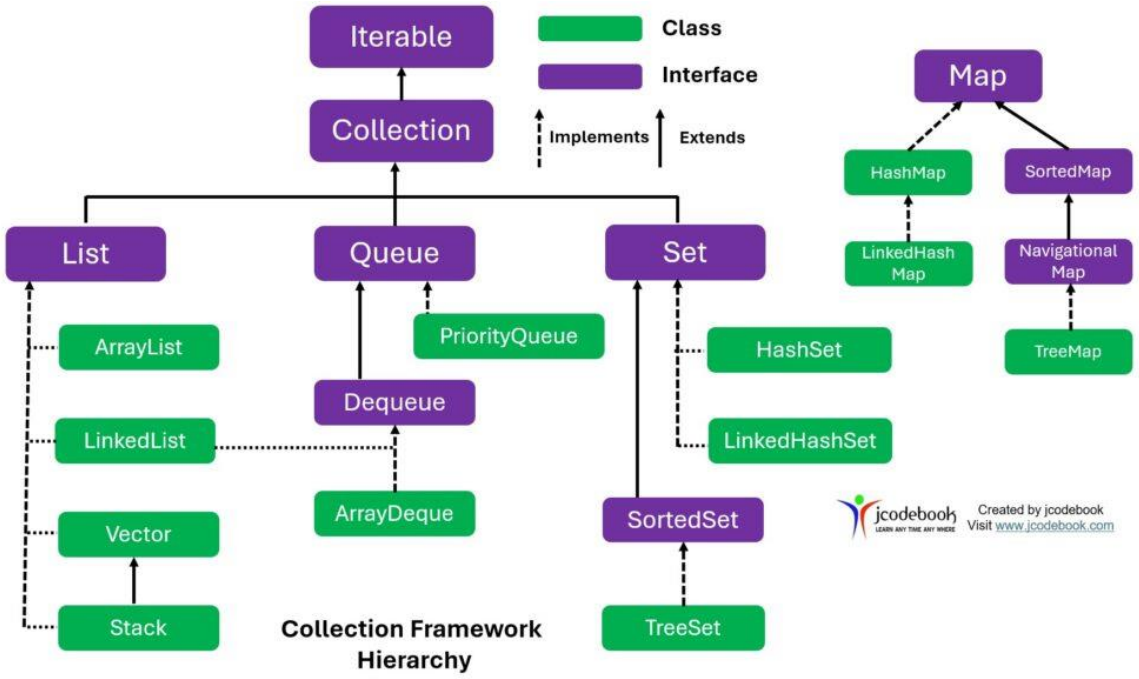
List: **An ordered collection that can contain duplicate elements (e.g., ArrayList, LinkedList).**

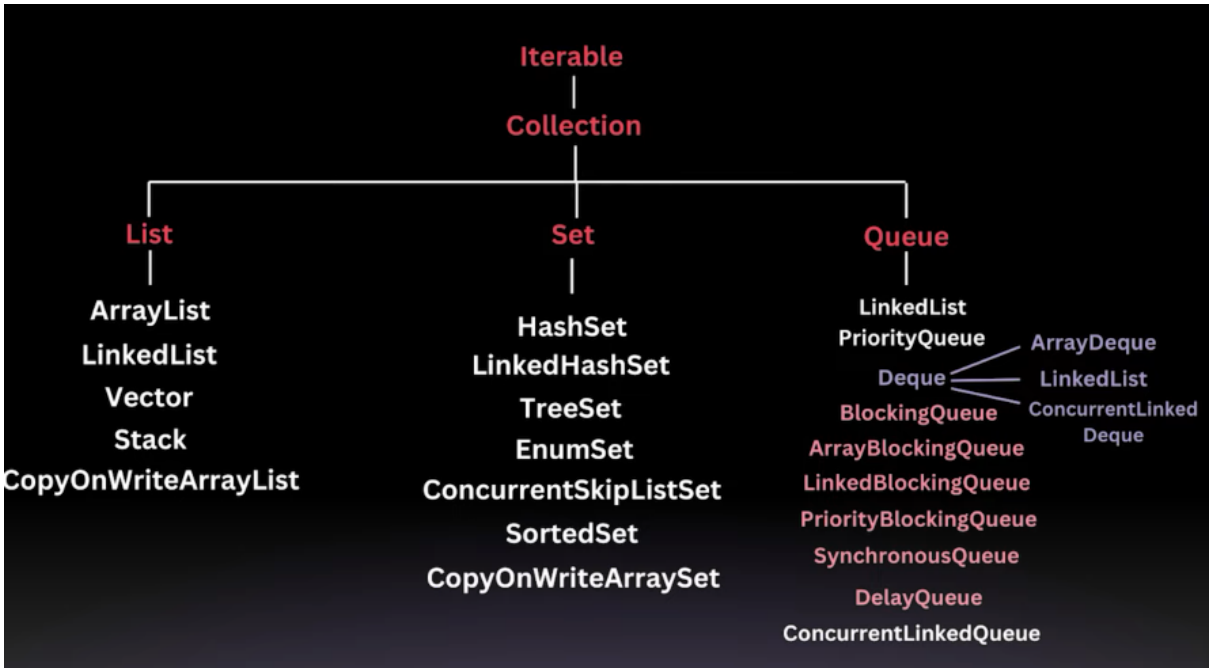
Set: **A collection that cannot contain duplicate elements (e.g., HashSet, TreeSet).**

Queue: **A collection designed for holding elements prior to processing (e.g., PriorityQueue, LinkedList when used as a queue).**

Deque: **A double-ended queue that allows insertion and removal from both ends (e.g., ArrayDeque).**

Map: **An interface that represents a collection of key-value pairs (e.g., HashMap, TreeMap).**





For-each loop syntax works on arrays and anything that implements Iterable.

forEach() method works on Iterable (e.g., ArrayList, Set, etc.) and **Streams**.

```
for (String item : list) {
    System.out.println(item);
}

list.forEach(item -> System.out.println(item));
```

Since Collection is an interface, it cannot be instantiated directly; rather, it provides a blueprint for the basic operations that are common to all collections.

The Collection interface defines a set of core methods that are implemented by all classes that implement the interface. These methods allow for basic operations such as adding, removing, and checking the existence of elements in a collection.

List Interface

The List interface in Java is a part of the java.util package and is a sub-interface of the Collection interface. It provides a way to store an *ordered collection of elements* (known as a sequence). Lists allow for precise control over where elements are inserted and *can contain duplicate elements.*

Key Features of the List Interface

Order Preservation
Index-Based Access
Allows Duplicates

ArrayList

An ArrayList is a resizable array implementation of the List interface. Unlike arrays in Java, which have a fixed size, an *ArrayList can change its size dynamically* as elements are added or removed. This flexibility makes it a popular choice when the number of elements in a list isn't known in advance.

```
List<Integer> arrayList = new ArrayList<>();
ArrayList<Integer> list = new ArrayList<>();
list.add(10);  // at index 0
list.add(2);  // 1
list.add(55);  // 2
list.add(30);
list.add(23);
System.out.println(list.get(1)); // 2
System.out.println(list.size());

for(int i=0;i<list.size();i++){
    System.out.println(list.get(i));
}

for(int i:list){
    System.out.println(i);
}

System.out.println(list.contains(32));
System.out.println(list.contains(2));

list.remove( index: 2);
list.add( index: 2,  element: 60);
list.set(2, 50); // it will replace the value
System.out.println(list); // it will use toString() method implementation to print
```

Internal working

Unlike a regular array, which has a fixed size, an ArrayList can grow and shrink as elements are added or removed. This dynamic resizing is achieved by creating a new array when the current array is full and copying the elements to the new array.

Internally, the ArrayList is implemented as an array of Object references. When you add elements to an ArrayList, you're essentially storing these elements in this internal array.

When you create an ArrayList, it has an initial capacity (default is 10). The capacity refers to the size of the internal array that can hold elements before needing to resize.

Adding Elements

When we add an element to an ArrayList, the following steps occur

Check Capacity: Before adding the new element, ArrayList checks if there is enough space in the internal array (elementData). If the array is full, it needs to be resized.

Resize if Necessary: If the internal array is full, the ArrayList will create a new array with a larger capacity (usually 1.5 times the current capacity) and copy the elements from the old array to the new array.

Add the Element: The new element is then added to the internal array at the appropriate index, and the size is incremented.

Resizing the Array

- *Initial Capacity:* By default, the initial capacity is 10. This means the internal array can hold 10 elements before it needs to grow.
- *Growth Factor:* When the internal array is full, a new array is created with a size 1.5 times the old array. This growth factor balances memory efficiency and resizing cost.
- *Copying Elements:* When resizing occurs, all elements from the old array are copied to the new array, which is an O(n) operation, where n is the number of elements in the ArrayList.

> The array buffer into which the elements of the ArrayList are stored. The capacity of the ArrayList is the length of this array buffer. Any empty ArrayList with elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA will be expanded to DEFAULT_CAPACITY when the first element is added.

```
transient Object[] elementData; // non-private to simplify nested class access
```

Removing Elements

- *Check Bounds:* The ArrayList first checks if the index is within the valid range.
- *Remove the Element*: The element is removed, and all elements to the right of the removed element are shifted one position to the left to fill the gap.
- *Reduce Size:* The size is decremented by 1.

Note :-

To prevent the overhead of copying elements, if u know that ArrayList will have alteast n no of element then we can give initial capacity at initialisation.

```java
List<Integer> list = new ArrayList<>( initialCapacity: 1000);
System.out.println(list.size());  // 0
System.out.println(list.get(0));  // IndexOutOfBoundsException
```

```java
List<Integer> list = new ArrayList<>( initialCapacity: 11);
System.out.println(list.size());  // 0

// printing capacity using Reflection

Field field = ArrayList.class.getDeclaredField( name: "elementData");
field.setAccessible(true);
Object[] elementData = (Object[]) field.get(list);
System.out.println("Arraylist capacity: " + elementData.length); // 11
```

```java
List<Object> list = new ArrayList<>();
System.out.println(list.getClass().getName());

// other ways of creating list
List<String> list1 = Arrays.asList("monday", "tuesday"); // return fixed size
System.out.println(list1.getClass().getName());
//  list1.add("friday"); // exception, we cannot add elements but can remove/replace
list1.set(1, "saturday");

String[] array = {"Arjun", "Aakash", "Ishan"};
List<String> list2 = Arrays.asList(array);
System.out.println(list2.getClass().getName());
```

```
Main ×
"C:\Program Files\Java\jdk-17\bin\java.exe" --add-opens java.base/java.util=ALL-UNNAMED "-javaa
java.util.ArrayList
java.util.Arrays$ArrayList
java.util.Arrays$ArrayList
```

In ArrayList constructor we can give capacity or Collection.

```java
// other ways of creating list
List<String> list1 = Arrays.asList("monday", "tuesday"); // return fixed size

List<String> list4 = new ArrayList<>(list1);
list4.add("sunday");  // allowed

List<Integer> integers = List.of(1, 2, 3); // unmodifiable list
```

```java
List<Integer> list = new ArrayList<>();
list.add(10);
List<Integer> list1 = List.of(1, 2, 3, 4, 5, 6);
list.addAll(list1);
System.out.println(list);  // [10, 1, 2, 3, 4, 5, 6]
```

```java
// Removing by index
list.remove(1); // Removes the element at index 1 ("Grapes")

// Removing by value
list.remove("Apple"); // Removes "Apple" from the list
```

```java
List<String> fruits = new ArrayList<>();
fruits.add("Apple");
fruits.add("Mango");
fruits.add("Apple");
fruits.add("Cherry");

fruits.remove( o: "Apple"); // remove the first occurrence
System.out.println(fruits);

List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);
list.remove( index: 1);  // this will remove index not the value 1
list.remove(Integer.valueOf( i: 1)); // it will remove first occurrence of 1
```

```java
boolean hasApple = list.contains("Apple"); // Returns true if "Apple" is present
```

Converting to Array

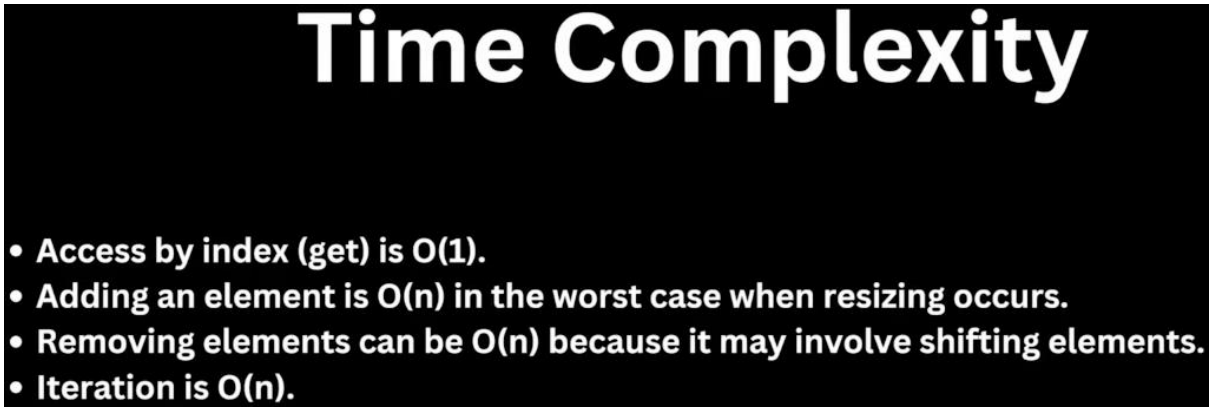
```java
String[] array = list.toArray(new String[0]);
```

```java
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
list.add(3);

Integer[] array = list.toArray(new Integer[0]);
System.out.println(array);
```

Sorting an ArrayList

```java
Collections.sort(list); // Sorts in natural order
```

```java
List<Integer> list = new ArrayList<>();
list.add(10);
list.add(2);
list.add(3);

Collections.sort(list);
list.sort( c: null);  // same thing, just that we passed null for comparator
System.out.println(list);
```



Comparator

```
@FunctionalInterface
public interface Comparator<T> {
```

Its an interface using which we can do custom sorting.

It has `int compare(T o1, T o2);` which compares two objects of same type and determine their order.

The integer returned tells the relative order of two objects.

- If -ve then o1 comes first then o2
- 0 then o1 & o2 equal in ordering
- +ve then o2 comes first then o1

```java
class MyComparator implements Comparator<Integer>{

    @Override
    public int compare(Integer o1, Integer o2) {
        return o2 - o1; // descending order
    }
}
// 5  3
// 3  5
public class Main {
    public static void main(String[] args) throws Exception {
        List<Integer> list = new ArrayList<>();
        list.add(3);
        list.add(1);
        list.add(2);
        list.sort(new MyComparator());
        System.out.println(list); // [3, 2, 1]
```

```java
class StringLengthComparator implements Comparator<String>{

    @Override
    public int compare(String o1, String o2) {
        return o1.length() - o2.length();
    }
}
// "ok"  "bye"
// "bye"  "ok"

public class Main {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("banana", "apple", "date");

        // I want to sort based on the size of string
        words.sort(new StringLengthComparator());
        System.out.println(words); // [date, apple, banana]
```

```java
List<String> words = Arrays.asList("banana", "apple", "date");
words.sort((a, b) -> a.length() - b.length());
System.out.println(words); // [date, apple, banana]

List<Integer> list = new ArrayList<>();
list.add(3);
list.add(1);
list.add(2);
list.sort((a, b) -> b - a);
System.out.println(list); // [3, 2, 1]
```

```java
class Student{
    private String name;
    private double gpa;

    public String getName() {return name;}
    public double getGpa() {return gpa;}

    public Student(String name, double gpa) {
        this.name = name;
        this.gpa = gpa;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Student> students = new ArrayList<>();
        students.add(new Student( name: "Charle",  gpa: 3.5));
        students.add(new Student( name: "Bob",  gpa: 3.7));
        students.add(new Student( name: "Alice",  gpa: 3.5));
        students.add(new Student( name: "Akshit",  gpa: 3.9));
```

```java
// sort in decreasing order of gpa but ascending order of names
students.sort((a, b) -> {
    if(b.getGpa() - a.getGpa() > 0) return 1;
    else if(b.getGpa()  - a.getGpa() < 0) return -1;
    else {
        // compareTo is method of String
        // return 0 if same, -ve if a.getName() comes first in dictionary, +ve if it comes after b.getName()
        return a.getName().compareTo(b.getName());
    }
});

// using Java 8 method reference
Comparator<Student> comparator = Comparator.comparing(Student::getGpa).reversed()
                                        .thenComparing(Student::getName);
students.sort(comparator);
// or
Collections.sort(students, comparator);

for(Student s: students){
    System.out.println(s.getName() + " : " + s.getGpa());
}
// Akshit : 3.9
// Bob : 3.7
// Alice : 3.5
// Charle : 3.5
```

```java
List<Student> students = Arrays.asList(
    new Student("Alice", 22),
    new Student("Bob", 22),
    new Student("Charlie", 20)
);

students.sort(
    Comparator
        .comparing((Student s) -> s.getAge())
        .reversed()
        .thenComparing((Student s) -> s.getName())
);
```

This sorts:

1. By **age descending**
2. If ages are equal, by **name ascending**

Sort List of String in reverse lexicographically order –

✅ Example: Using `List.sort()` (Java 8+)

```java
List<String> words = Arrays.asList("banana", "apple", "grape", "cherry");
words.sort(Comparator.reverseOrder());
System.out.println(words);
```

✅ Example: Custom Comparator (manual)

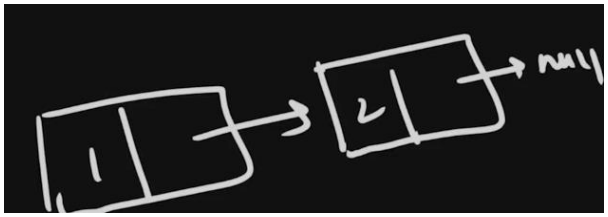
```java
words.sort((a, b) -> b.compareTo(a)); // reverse of a.compareTo(b)
```

LinkedList

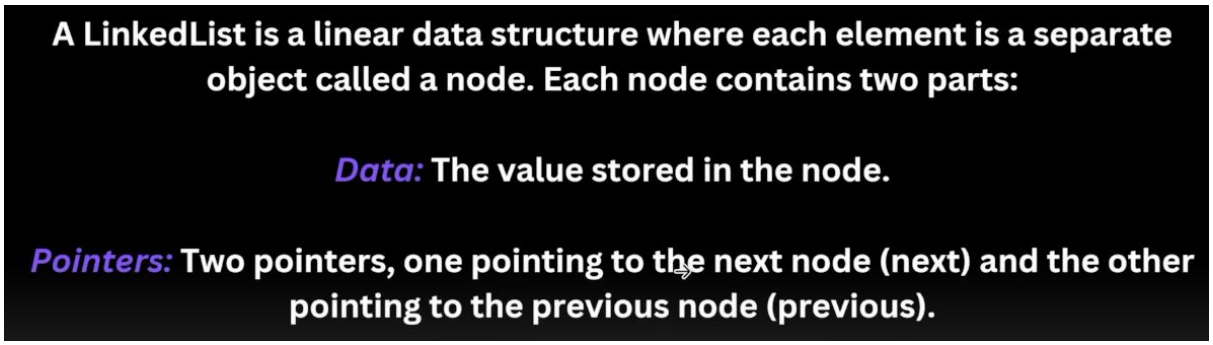
The LinkedList class in Java is a part of the Collection framework and implements the List interface. Unlike an ArrayList, which uses a dynamic array to store the elements, *a LinkedList stores its elements as nodes in a doubly linked list*. This provides different performance characteristics and usage scenarios compared to ArrayList.

Custom implementation will be like –

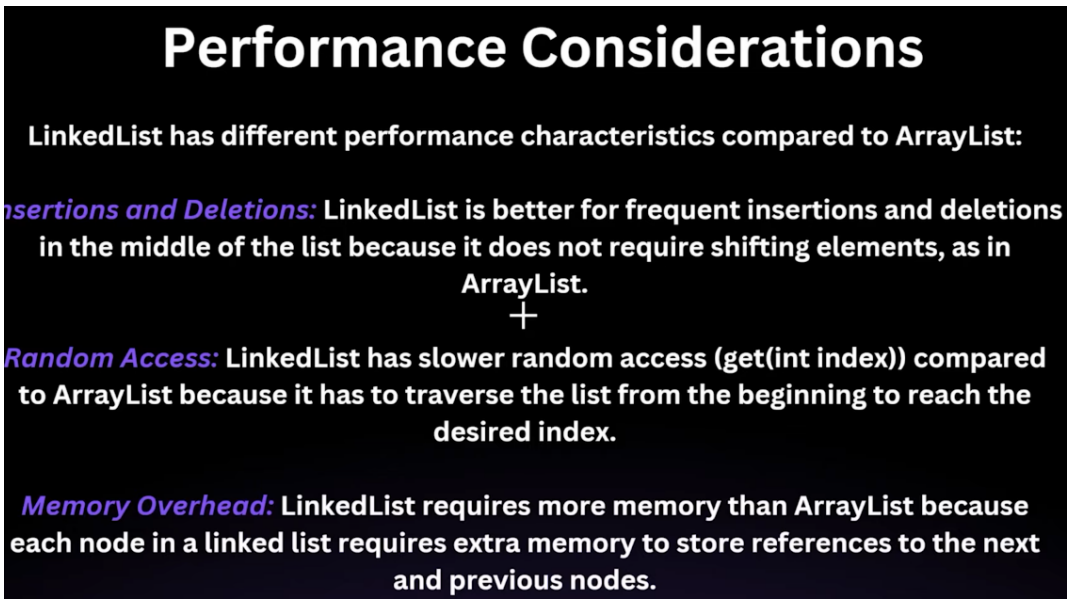
```java
class Node{
    public int value;
    public Node next; // its reference to next node
}

public class Test {
    public static void main(String[] args) {
        Node n1 = new Node();
        Node n2 = new Node();
        n2.value = 2;
        n1.value = 1;
        n1.next = n2;
        n2.next = null;
```



But Java internally provides –



We can't directly access the elements with .get(i) in O(1) instead we have to run loop so time complexity of get(i) is O(n)

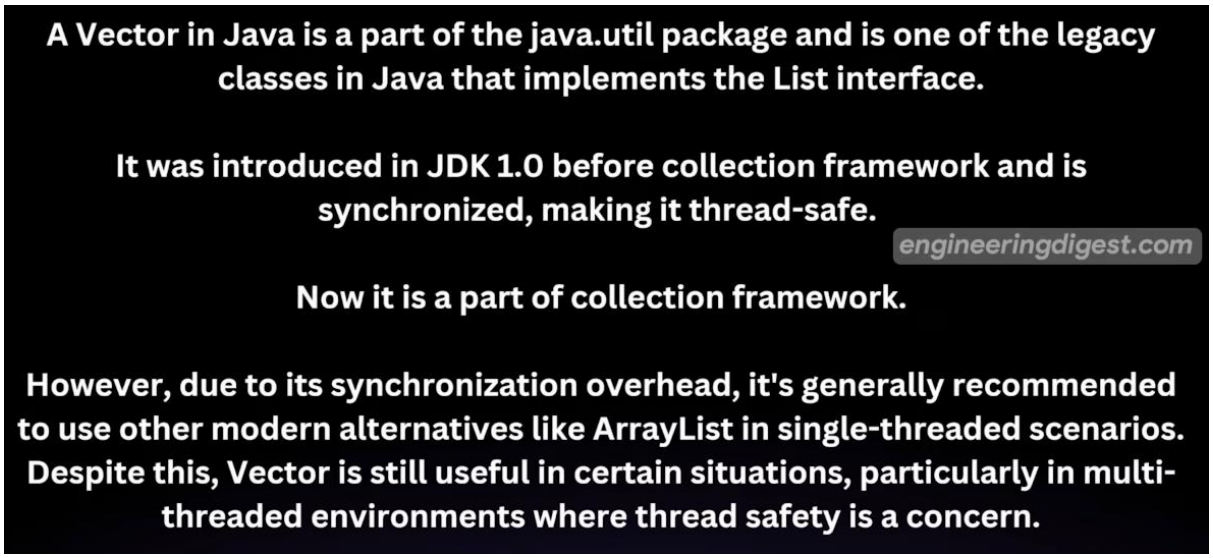


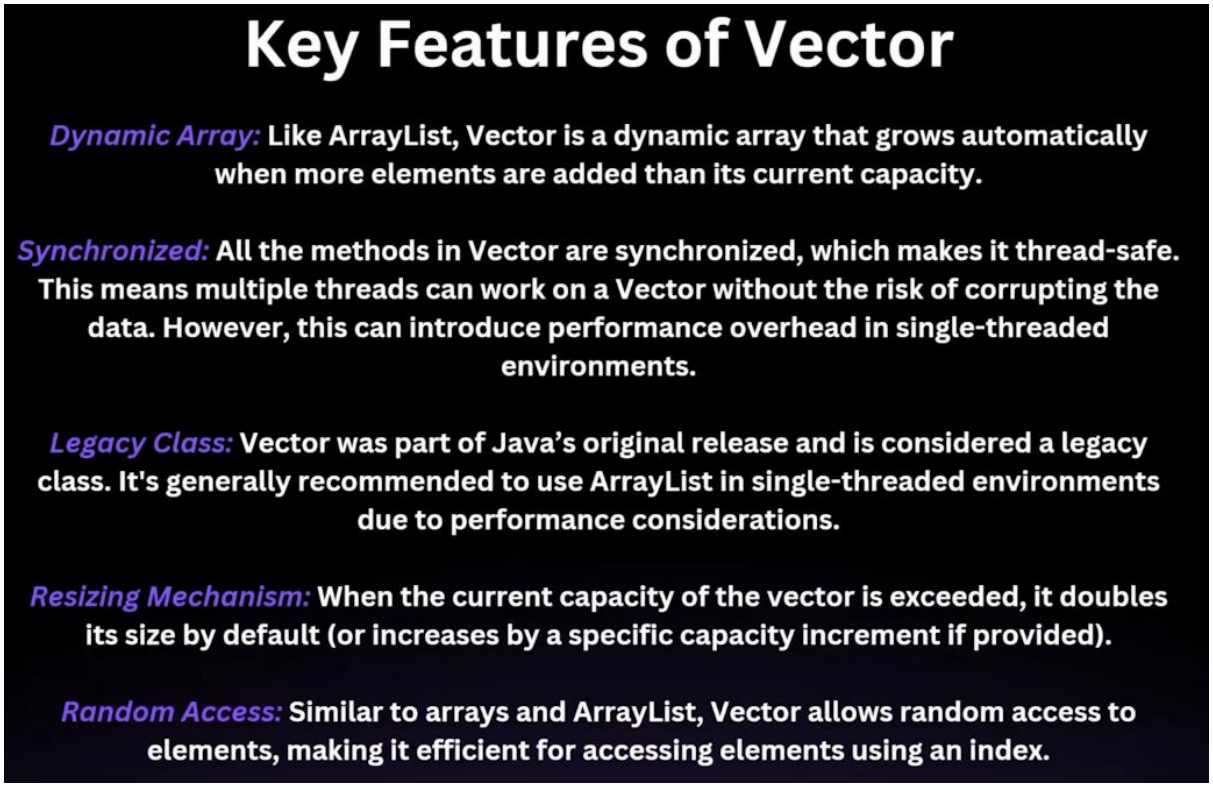

```java
LinkedList<Integer> linkedList = new LinkedList<>();
linkedList.add(1);
linkedList.add(2);
linkedList.add(3);
linkedList.get(2); // O(n)
linkedList.addLast( e: 4); // O(1)
linkedList.addFirst( e: 0); // O(1)
linkedList.add( index: 2,  element: 10); // O(n)
linkedList.getFirst();
linkedList.getLast();
System.out.println(linkedList); // [0, 1, 10, 2, 3, 4]
linkedList.removeIf(x -> x%2==0);
System.out.println(linkedList); // [1, 3]

LinkedList<String> animals = new LinkedList<>(Arrays.asList("Cat", "Dog", "Monkey"));
LinkedList<String> animalsToRemove = new LinkedList<>(Arrays.asList("Dog", "Lion"));
animals.removeAll(animalsToRemove);
System.out.println(animals); // [Cat, Monkey]
```

We can also write reference as List, but then we won't be able to access the method which aren't part of List interface.

```java
List<Integer> linkedList = new LinkedList<>();
linkedList.add(1);
linkedList.add(2);
linkedList.add(3);
linkedList.get(2); // O(n)
linkedList.addLast(4); // O(1)
linkedList.addFirst(0); // O(1)
```

Vector





Constructors of Vector

Vector(): Creates a vector with an initial capacity of 10.

Vector(int initialCapacity): Creates a vector with a specified initial capacity.

Vector(int initialCapacity, int capacityIncrement): Creates a vector with an initial capacity and capacity increment (how much the vector should grow when its capacity is exceeded).

Vector(Collection<? extends E> c): Creates a vector containing the elements of the specified collection.

```
Vector<Integer> vector = new Vector<>();
System.out.println(vector.capacity()); // 10
```

```
Vector<Integer> vector = new Vector<>( initialCapacity: 5);
vector.add(1);
vector.add(1);
vector.add(1);
vector.add(1);
vector.add(1);
vector.add(1);
System.out.println(vector.capacity()); // 10, it becomes doubled when exceed
```

```
Vector<Integer> vector = new Vector<>( initialCapacity: 5,  capacityIncrement: 3);
vector.add(1);
vector.add(1);
vector.add(1);
vector.add(1);
vector.add(1);
System.out.println(vector.capacity()); // 5
vector.add(1);
System.out.println(vector.capacity()); // 8
```

```
LinkedList<Integer> linkedList = new LinkedList<>();
linkedList.add(1);
linkedList.add(2);
linkedList.add(3);
Vector<Integer> vector = new Vector<>(linkedList);
System.out.println(vector); // [1, 2, 3]
```

Methods in Vector

- *add(E e):* Adds an element at the end.
- *add(int index, E element):* Inserts an element at the specified index.
- *get(int index):* Retrieves the element at the specified index.
- *set(int index, E element):* Replaces the element at the specified index.
- *remove(Object o):* Removes the first occurrence of the specified element.
- *remove(int index):* Removes the element at the specified index.
- *size():* Returns the number of elements in the vector.
- *isEmpty():* Checks if the vector is empty.
- *contains(Object o):* Checks if the vector contains the specified element.
- *clear():* Removes all elements from the vector.

Internal Implementation of Vector

Internally, Vector uses an array to store its elements. The size of this array grows as needed when more elements are added. The default behavior is to double the size of the array when it runs out of space. This resizing operation is a costly one, as it requires copying the old elements to the new, larger array.

Synchronization and Performance

Since Vector methods are synchronized, it ensures that only one thread can access the vector at a time. This makes it thread-safe but can introduce performance overhead in single-threaded environments because synchronization adds locking and unlocking costs.

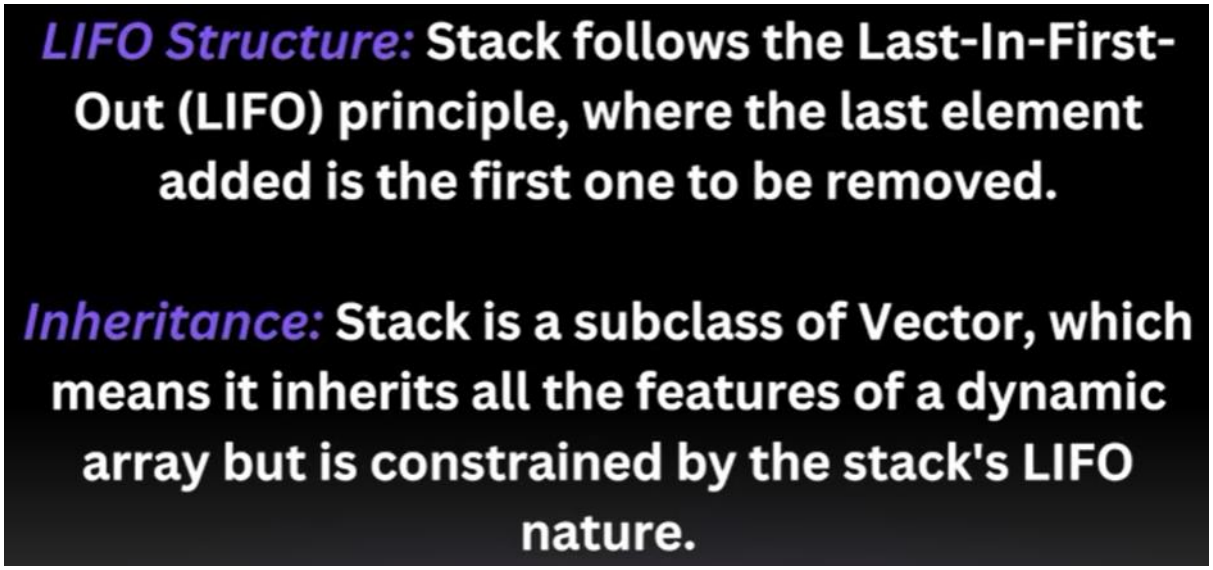
In modern Java applications, ArrayList is generally preferred over Vector when synchronization isn't required. For thread-safe collections, the CopyOnWriteArrayList or ConcurrentHashMap from the java.util.concurrent package is often recommended instead.

```java
ArrayList<Integer> list = new ArrayList<>();
Thread t1 =  new Thread(()->{
    for(int i=0;i<1000;i++){
        list.add(i);
    }
});
Thread t2 =  new Thread(()->{
    for(int i=0;i<1000;i++){
        list.add(i);
    }
});

t1.start();
t2.start();

try{
    t1.join();
    t2.join();
}catch (InterruptedException e){
    e.printStackTrace();
}

System.out.println(list.size());  // smaller than 2000
```

Threadsafe – at same instant only 1 thread can access the list and add

```java
Vector<Integer> list = new Vector<>();
Thread t1 =  new Thread(()->{...});
Thread t2 =  new Thread(()->{...});
t1.start();
t2.start();
try{
    t1.join();
    t2.join();
}catch (InterruptedException e){
    e.printStackTrace();
}
System.out.println(list.size());  // 2000
```

Stack




```java
Stack<Integer> stack = new Stack<>();
stack.push( item: 1);
stack.push( item: 2);
stack.push( item: 3);
stack.push( item: 4);
System.out.println(stack); // [1, 2, 3, 4]
Integer removedElement = stack.pop(); // 4
System.out.println(stack); // [1, 2, 3]
Integer peek = stack.peek(); // 3
System.out.println(stack); // [1, 2, 3]

System.out.println(stack.empty()); // false
System.out.println(stack.size()); // 3

int idx = stack.search( o: 3);  // it gives 1 based indexing from top
System.out.println(idx); // 1
```

LinkedList as stack : -

```
LinkedList<Integer> linkedList = new LinkedList<>();
linkedList.addLast( e: 1);
linkedList.addLast( e: 2);
linkedList.addLast( e: 3);
linkedList.getLast();
linkedList.removeLast();
linkedList.size();
linkedList.isEmpty();
```

In single threaded environment we can use LinkedList as stack to avoid synchronization overheads.

ArrayList as stack :-

```
ArrayList<Integer> arrayList = new ArrayList<>();
arrayList.add(1);
arrayList.add(2);
arrayList.add(3);
arrayList.get(arrayList.size() - 1); // peek
arrayList.remove( index: arrayList.size() -1); // pop
```

CopyOnWriteArrayList

"Copy on Write" means that whenever a write operation like adding or removing an element then instead of directly modifying the existing list, a new copy of the list is created, and the modification is applied to that copy. This ensures that other threads reading the list while it's being modified are unaffected.

Read Operations: Fast and direct, since they happen on a stable list without interference from modifications.

Write Operations: A new copy of the list is created for every modification. The reference to the list is then updated so that subsequent reads use this new list.

This is preferred when reads are more and write are less becoz creating new copy of list needs memory consumption.

It is thread safe.

ArrayList should not get modified while reading -

```
List<String> shoppingList = new ArrayList<>();
shoppingList.add("Milk");
shoppingList.add("Eggs");
shoppingList.add("Bread");

for(String item : shoppingList){
    if(item.equals("Eggs")){
        shoppingList.add("Butter"); // will get ConcurrentModificationException
        System.out.println("Added butter while reading");
    }
}
```

Works fine now -

```java
public class Test {
    public static void main(String[] args) {
        List<String> shoppingList = new CopyOnWriteArrayList<>();
        shoppingList.add("Milk");
        shoppingList.add("Bread");
        shoppingList.add("Eggs");

        for(String item : shoppingList){
            System.out.println(item);
            if(item.equals("Eggs")){
                shoppingList.add("Butter");
                System.out.println("Added butter while reading");
            }
        }
        System.out.println("Updated shopping list :" + shoppingList);
    }
}
```

```
Test
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\J
Milk
Bread
Eggs
Added butter while reading
Updated shopping list :[Milk, Bread, Eggs, Butter]
```

Reading and modifying happening at same time. Its possible because read happened on original snapshot but write happened on separate copy. Loop will only run on original list. When read is complete then shoppingList reference variable will start pointing to new copy list.

Another example to demonstrate –

```java
List<Integer> sharedList = new CopyOnWriteArrayList<>();
sharedList.add(1);
sharedList.add(1);
sharedList.add(1);

Thread readerThread = new Thread(()->{
    while(true){
        for(Integer i:sharedList){
            System.out.println(i);
        }
    }
});

Thread writerThread = new Thread(()->{
    try {
        sharedList.add(4);
        Thread.sleep( millis: 500);
        sharedList.add(5);
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
});
readerThread.start();
writerThread.start();
```

Map

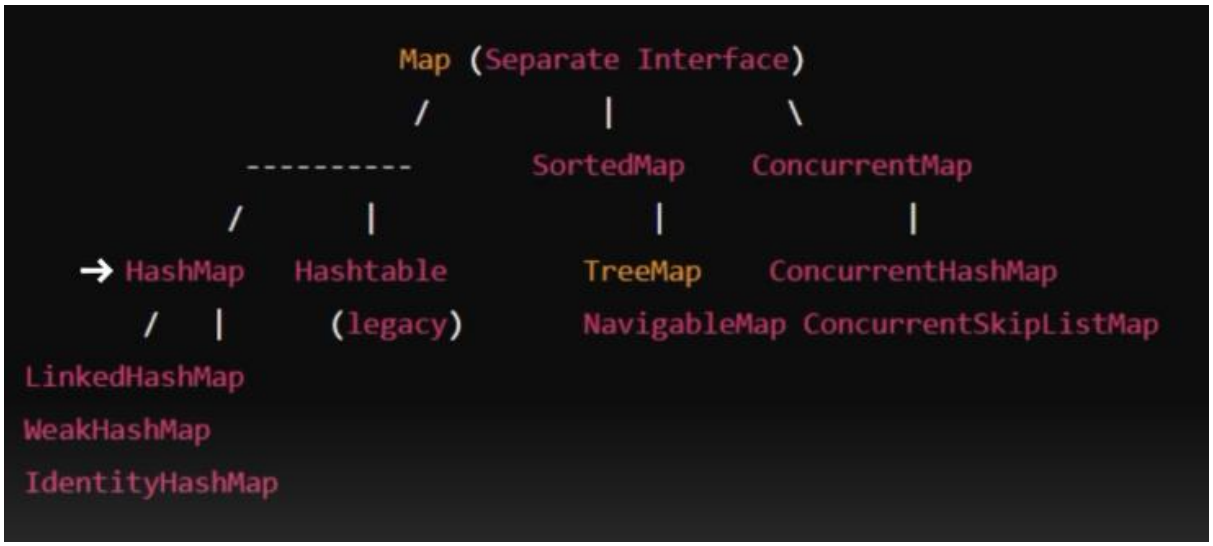
In Java, a Map is an object that maps keys to values. It cannot contain duplicate keys, and each key can map to at most one value. *Think of it as a dictionary* where you look up a word (key) to find its definition (value).

Map does not extend the Collection interface

Key Characteristics of the Map Interface

Key-Value Pairs: Each entry in a Map consists of a key and a value.
Unique Keys: No two entries can have the same key.
One Value per Key: Each key maps to a single value.
Order: Some implementations maintain insertion order (LinkedHashMap), natural order (TreeMap), or no order (HashMap).

HashMap



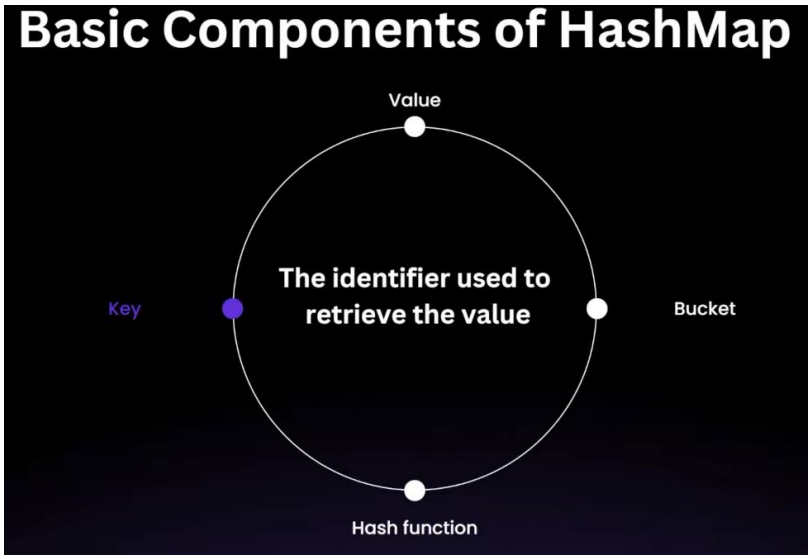
```java
HashMap<Integer,String> map = new HashMap<>();
map.put(1, "Akshit");
map.put(3, "Akash");
map.put(2, "Arjun");
System.out.println(map); // {1=Akshit, 2=Arjun, 3=Akash}

String s1 = map.get(3);
System.out.println(s1); // Akash
String s2 = map.get(70);
System.out.println(s2); // null

System.out.println(map.containsKey(2)); // true
System.out.println(map.containsValue("Akash")); // true

// traversal
// Set<Integer> keys = map.keySet();
for(int i : map.keySet()){
    System.out.println(map.get(i));
}

// Set<Map.Entry<Integer, String>> entries = map.entrySet();
for(Map.Entry<Integer, String> entry : map.entrySet()){
    entry.setValue(entry.getValue().toUpperCase());
    System.out.println(entry.getKey() + ": " + entry.getValue());
}
```

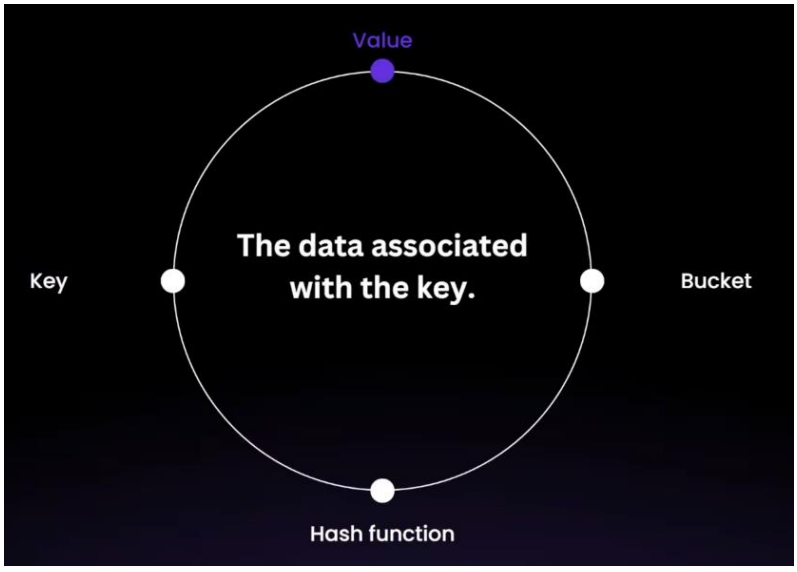
Key Characteristics

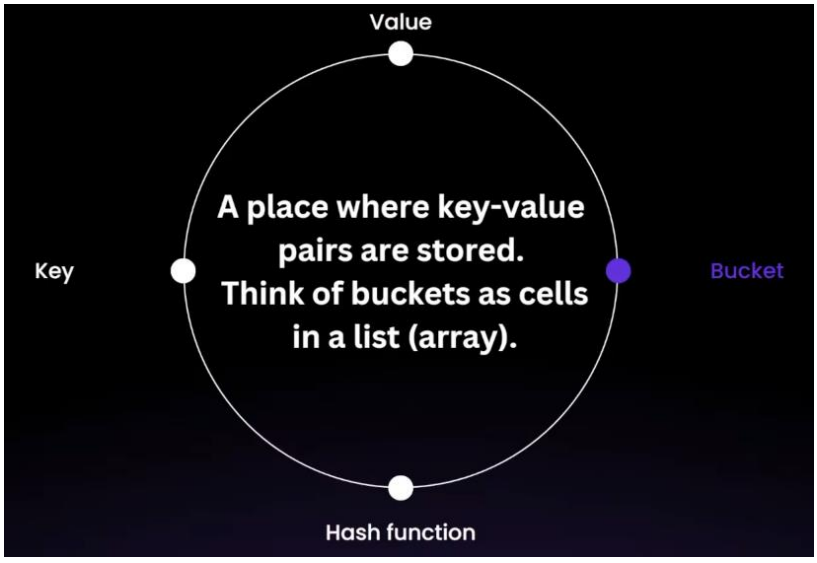
Unordered: **Does not maintain any order of its elements.**
Allows null Keys and Values: **Can have one null key and multiple null values.**
Not Synchronized: **Not thread-safe; requires external synchronization if used in a multi-threaded context.**
Performance: **Offers constant-time performance (O(1)) for basic operations like get and put, assuming the hash function disperses elements properly.**

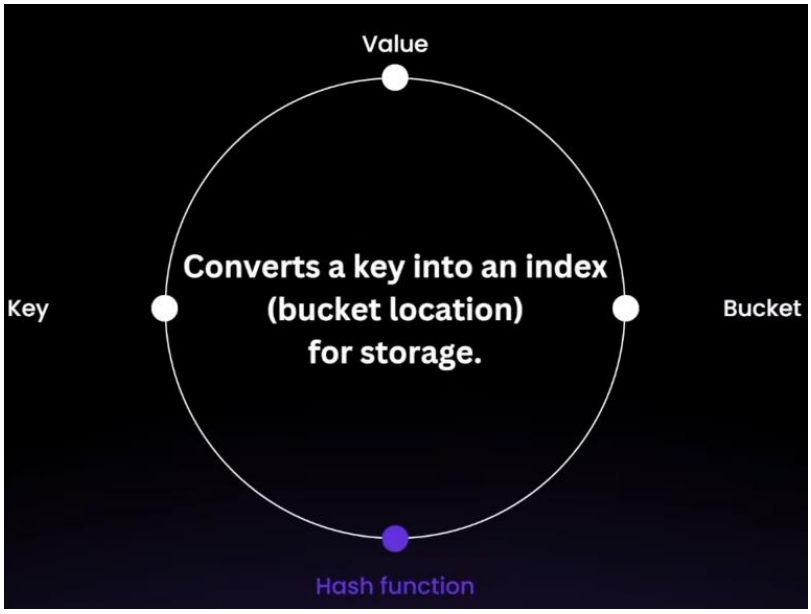
```java
HashMap<Integer,String> map = new HashMap<>();
map.put(1, "Akshit");
map.put(3, "Akash");
map.put(2, "Arjun");
map.remove( key: 3);
System.out.println(map); // {1=Akshit, 2=Arjun}

List<Integer> list = Arrays.asList(1,2,3,4,5,6);
System.out.println(list.contains(5)); // O(n) but map.containsKey(3) is O(1)
```

Internal Working of Hashmap









A hash function is an algorithm that takes an input (or "key") and returns a fixed-size string of bytes, typically a numerical value. The output is known as a hash code, hash value, or simply hash. *The primary purpose of a hash function is to map data of arbitrary size to data of fixed size*

- *Deterministic:* The same input will always produce the same output.
- *Fixed Output Size:* Regardless of the input size, the hash code has a consistent size (e.g., 32-bit, 64-bit).
- *Efficient Computation:* The hash function should compute the hash quickly.

How Data is Stored in HashMap

Step 1: Hashing the Key

First, the key is passed through a hash function to generate a unique hash code (an integer number). This hash code helps determine where the key-value pair will be stored in the array (called a "bucket array").

Step 2: Calculating the Index

The hash code is then used to calculate an index in the array (bucket location) using

int index = hashCode % arraySize;

The index decides which bucket will hold this key-value pair.

For example, if the array size is 16, the key's hash code will be divided by 16, and the remainder will be the index.

Step 3: Storing in the Bucket

The key-value pair is stored in the bucket at the calculated index. Each bucket can hold multiple key-value pairs

(this is called a collision handling mechanism, discussed later).

map.put("apple", 50);

- "apple" is the key.
- 50 is the value.
- The hash code of "apple" is calculated.
- The index is found using the hash code.
- The pair ("apple", 50) is stored in the corresponding bucket.

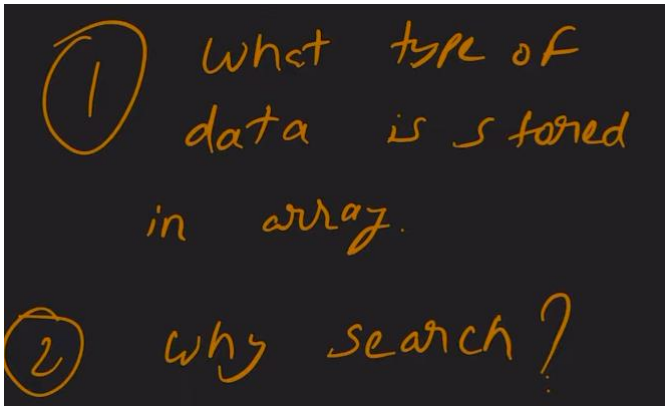
How HashMap Retrieves Data

When we call get(key), the HashMap follows these steps:

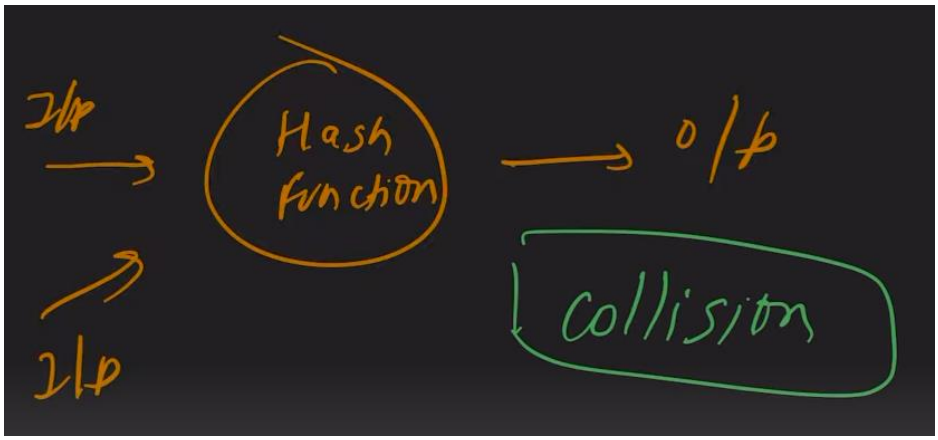
Hashing the Key: Similar to insertion, the key is hashed using the same hash function to calculate its hash code.

Finding the Index: The hash code is used to find the index of the bucket where the key-value pair is stored.

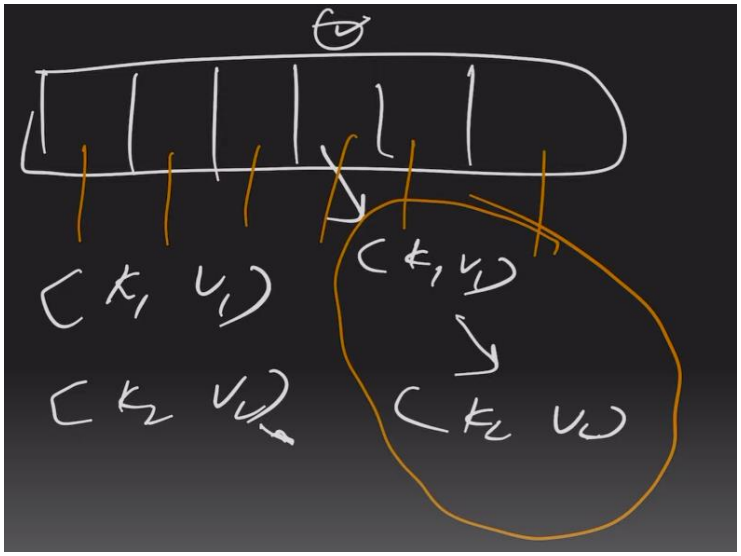
Searching in the Bucket: Once the correct bucket is found, it checks for the key in that bucket. If it finds the key, it returns the associated value.



Hash function give same output for different inputs leading to collision.



We do collision handling using separate chaining with linkedList. Bucket array is of type LinkedList



```
class Node<K, V> {
    final int hash;   // hash code of the key
    final K key;      // the key itself
    V value;          // the value associated with the key
    Node<K, V> next;  // pointer to the next node in case of a collision (linked list)
}
```

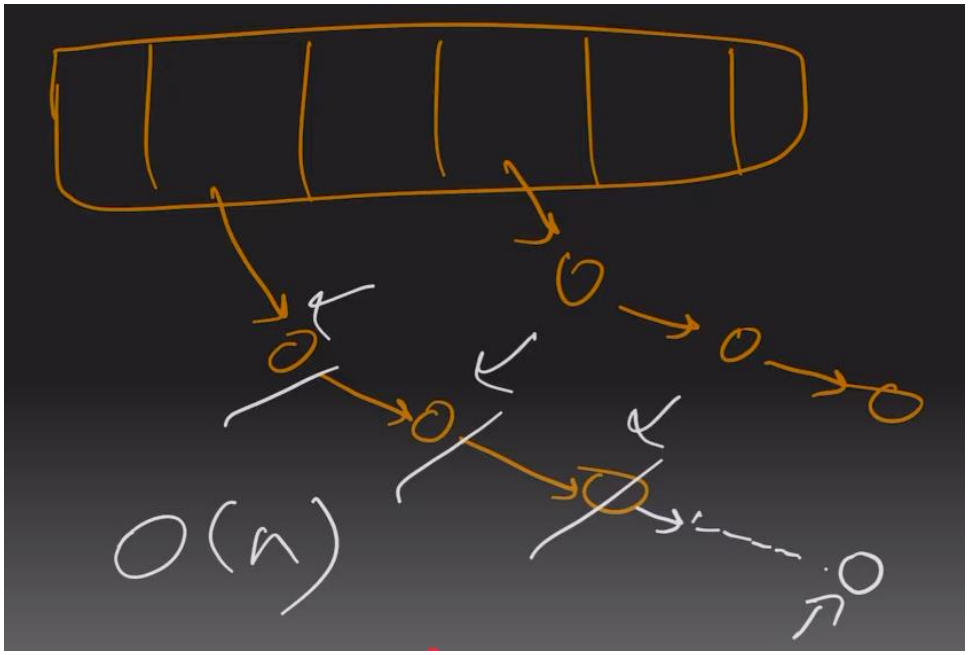
Handling Collisions

Since different keys can generate the same index (called a collision), HashMap uses a technique to handle this situation. Java's HashMap uses Linked Lists (or balanced trees after Java 8) for this.

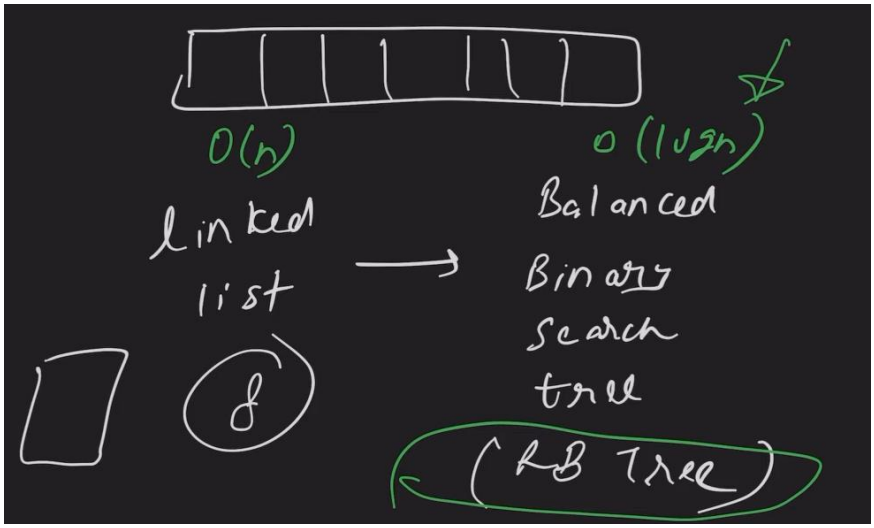
If multiple key-value pairs map to the same bucket, they are stored in a linked list inside the bucket.

When a key-value pair is retrieved, the HashMap traverses the linked list, checking each key until it finds a match.

In worst case time complexity is O(n) as searching in LinkedList is linear.



<u>After Java 8</u> :- When the linked list size crosses a certain threshhold, it is converted to Balanced Binary Search Tree (Red Black Tree). By default threshhold is 8.



Handling Collisions

map.put("apple", 50);
map.put("banana", 30);
map.put("orange", 80);

Let's say "apple" and "orange" end up in the same bucket due to a hash collision. They will be stored in a linked list in that bucket:

Bucket 5: ("apple", 50) -> ("orange", 80)

When we do map.get("orange"), HashMap will go to Bucket 5 and then traverse the linked list to find the entry with the key "orange".

HashMap Resizing (Rehashing)

HashMap has an internal array size, which by default is 16.
When the number of elements (key-value pairs) grows and exceeds a certain load factor (default is 0.75), HashMap automatically resizes the array to hold more data. This process is called rehashing.

The default size of the array is 16, so when more than 12 elements (16 * 0.75) are inserted, the HashMap will resize.

During rehashing

The array size is doubled.

1. **All existing entries are rehashed (i.e., their positions are recalculated) and placed into the new array.**
2. **This ensures the HashMap continues to perform efficiently even as more data is added.**

```
HashMap<Integer,String> map = new HashMap<>( initialCapacity: 17,  loadFactor: 0.5f);
```

Time Complexity

HashMap provides constant time O(1) performance for basic operations like put() and get() **(assuming no collisions). However, if there are many collisions, and many entries are stored in the same bucket, the performance can degrade to O(n), where n is the number of elements in that bucket.**

But after Java 8, if there are too many elements in a bucket, HashMap switches to a balanced tree instead of a linked list to ensure better performance O(log n).

Suppose we want to store information about the number of fruits in a store. Here's what we want to store:

Fruit	Quantity
Apple	50
Banana	30
Orange	80
Grape	20

```
HashMap<String, Integer> fruitMap = new HashMap<>();
```

Let's add the key-value pairs one by one.

```
fruitMap.put("Apple", 50);
```

Internal Process

The key "Apple" is hashed using its hashCode(). Let's assume "Apple" generates a hashCode of 10832233 (this is just an example value).

The hashCode is used to calculate the index in the internal array (bucket array). Let's say the array size is initially 16.

```
index = hashCode % arraySize;

index = 10832233 % 16 = 9;
```

This means "Apple" will be stored in bucket 9

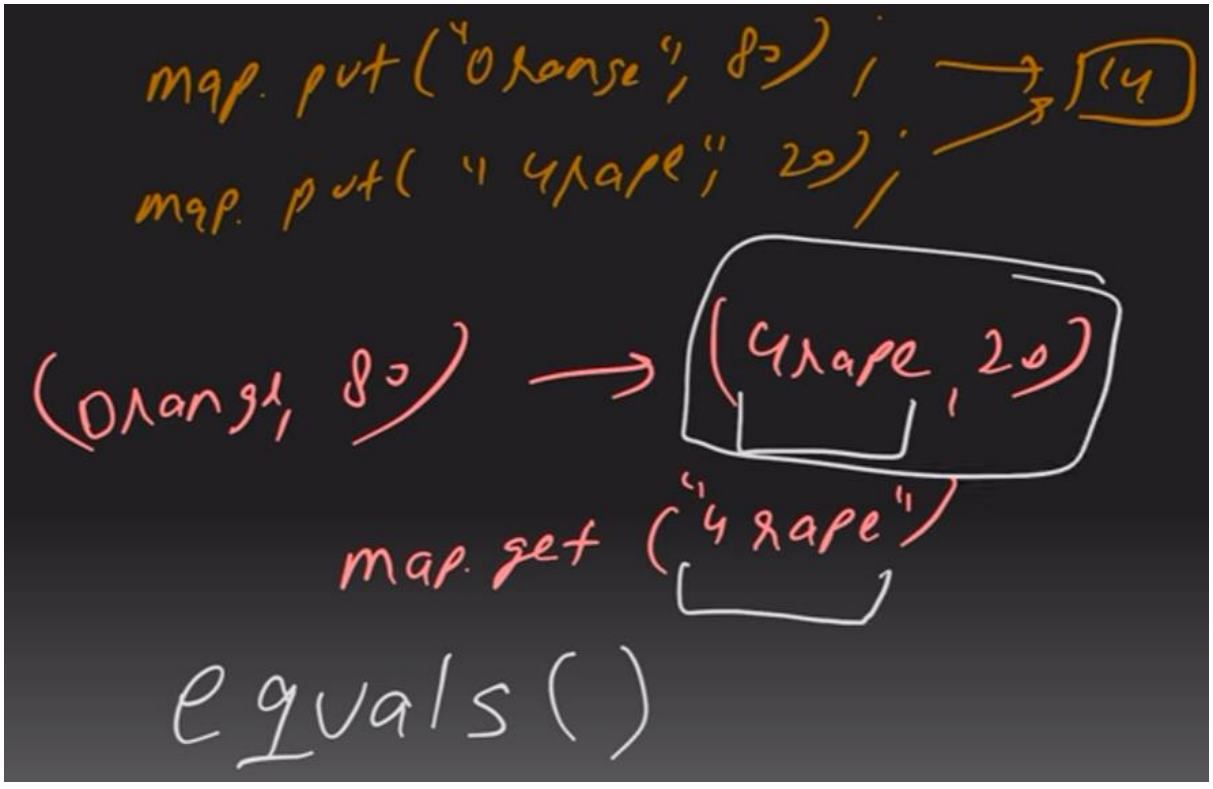
```
fruitMap.put("Banana", 30);
index = 13942244 % 16 = 4;

fruitMap.put("Grape", 20);
index = 548734 % 16 = 14;

fruitMap.put("Orange", 80);
index = 19332414 % 16 = 14;
```

```
fruitMap.put("Grape", 20);
index = 548734 % 16 = 14;

fruitMap.put("Orange", 80);
index = 19332414 % 16 = 14;
```

Since "Orange" is already stored in bucket 14, the HashMap handles the collision by adding "Grape" to the linked list in bucket 14.

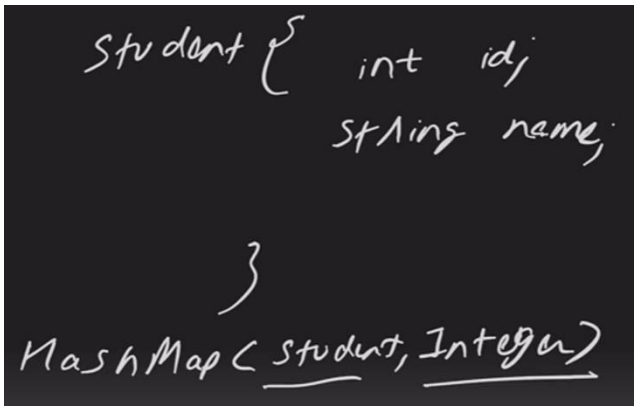
Now, bucket 14 contains two entries: ("Orange", 80) and ("Grape", 20).

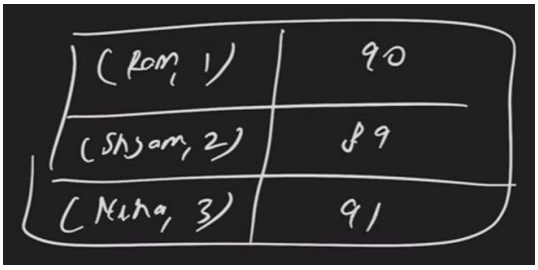
```
HashMap Structure (Array of Buckets, size: 16)

Index | Bucket (Key-Value Pairs)
-------------------------------------------------------------
  0   |
  1   |
  2   |
  3   |
  4   | ("Banana", 30)
  5   |
  6   |
  7   |
  8   |
  9   | ("Apple", 50)
 10   |
 11   |
 12   |
 13   |
 14   | ("Orange", 80) -> ("Grape", 20)   // Collision: stored in a linked list
 15   |
```

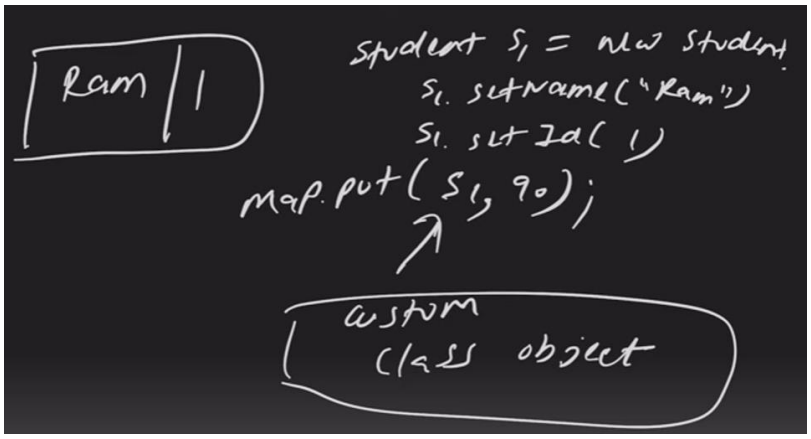


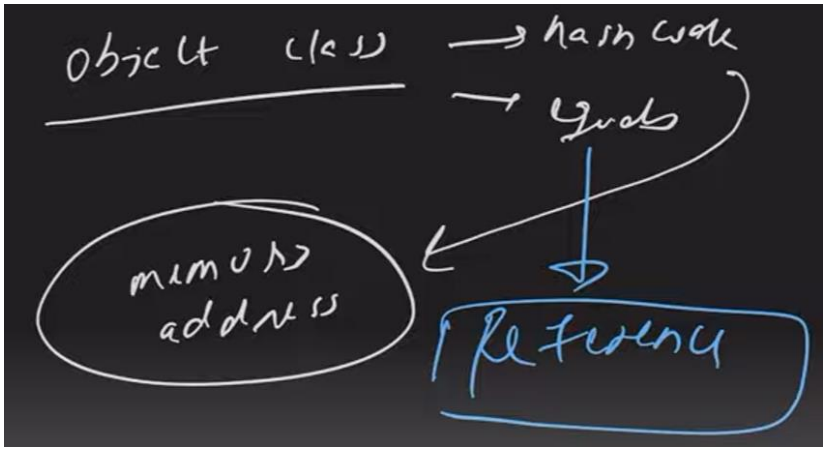
In above example, equals() method will be used to check String.

Another example –









Every instance of student class will have different hashcode based on memory address.

The default implementation of equals() in Object checks for **reference equality**, which means it compares the memory addresses of the two objects.

Ex - The String class overrides the equals method to compare the **content** of the strings rather than their memory addresses.

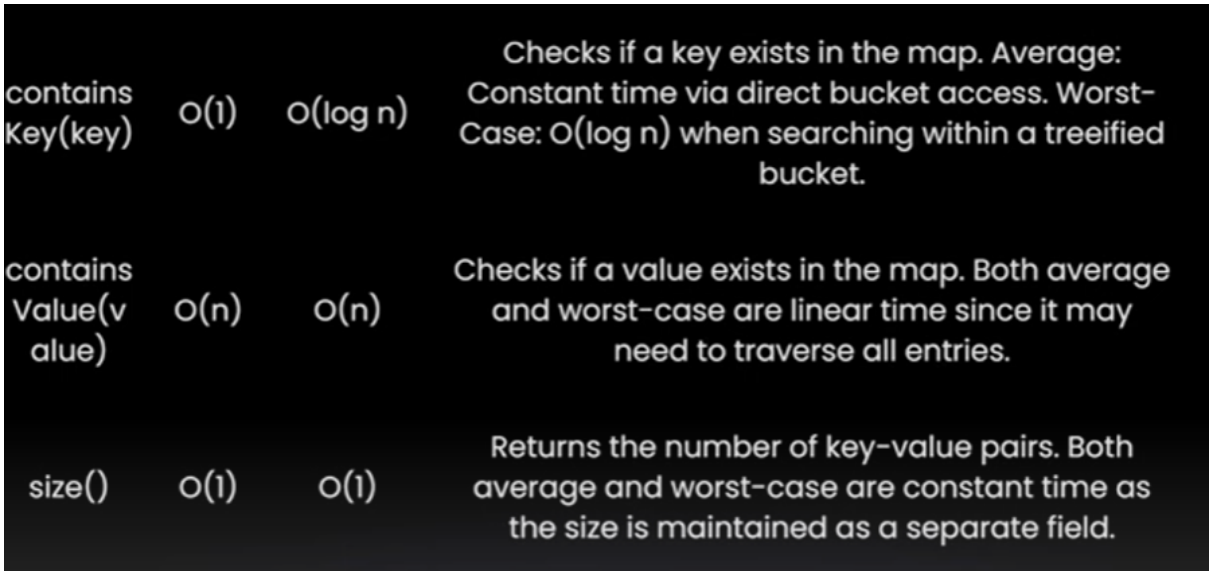
```java
class Person{
    private int id;
    private String name;
    public Person(String name, int id) {
        this.id = id;
        this.name = name;
    }
    public int getId() { return id; }
    public String getName() { return name; }
    public void setId(int id) { this.id = id; }
    public void setName(String name) { this.name = name; }
}
public class Test {
    public static void main(String[] args) {
        HashMap<Person, String> map = new HashMap<>();
        Person p1 = new Person( name: "Alice",  id: 1);
        Person p2 = new Person( name: "Bob",  id: 2);
        Person p3 = new Person( name: "Alice",  id: 1);
        // because p1 & p2 are created using "new" keyword they will be stored at different index
        // hashcode will be different becoz of different memory address

        map.put(p1, "Engineer"); // hashcode1 ----> index
        map.put(p2, "Designer"); // hashcode2 ----> index
        map.put(p3, "Manager"); // hashcode3 ----> index
        // total there will be 3 entries in map

        System.out.println(map.size()); // 3
        System.out.println(map.get(p1)); // Engineer
        System.out.println(map.get(p3)); // Manager

        Map<String, Integer> map1 = new HashMap<>();
        map1.put("Shubham", 90); // hashcode1 ---> index1
        map1.put("Neha", 92); // hashcode2 ---> index2
        map1.put("Shubham", 99); // hashcode1 ---> index1 ---> equals()  ---> replace value by 99
    }
```

We want to make it understand that p1 & p3 both are same and their hashcode generate should be same. To do that we have to override hashcode() and equals() method. We also override these methods when storing custom class objects in Set.

```java
class Person{
    private int id;
    private String name;
    public Person(String name, int id) {
        this.id = id;
        this.name = name;
    }
    public int getId() { return id; }
    public String getName() { return name; }
    public void setId(int id) { this.id = id; }
    public void setName(String name) { this.name = name; }

    @Override
    public int hashCode() {
        return Objects.hash(name, id);
    }

    @Override
    public boolean equals(Object obj) {
        if(this == obj) return true;
        if(this == null || getClass()!=obj.getClass()) return false;
        Person other = (Person) obj;
        return id == other.getId() && Objects.equals(name, other.getName());
    }

    @Override
    public String toString() {
        return "id :" + id + ", name: " + name;
    }
}
```

```java
HashMap<Person, String> map = new HashMap<>();
Person p1 = new Person( name: "Alice",  id: 1);
Person p2 = new Person( name: "Bob",  id: 2);
Person p3 = new Person( name: "Alice",  id: 1);

map.put(p1, "Engineer"); // hashcode1 ----> index1
map.put(p2, "Designer"); // hashcode2 ----> index
map.put(p3, "Manager"); // hashcode1 ----> index1 ----> equals() ----> replace "Engineer" by "Manager

System.out.println(map.size()); // 2
System.out.println(map.get(p1)); // Manager
System.out.println(map.get(p3)); // Manager
```

Operation	Average-Case Time Complexity	Worst-Case Time Complexity	Explanation
put(key, value)	O(1)	O(log n)	Inserts a key-value pair. Average: Constant time due to direct bucket access. Worst-Case: O(log n) when bucket converts to a Red-Black Tree after exceeding collision threshold.
get(key)	O(1)	O(log n)	Retrieves the value associated with a key. Average: Constant time via direct bucket access. Worst-Case: O(log n) when searching within a treeified bucket.
remove(key)	O(1)	O(log n)	Removes the key-value pair associated with a key. Average: Constant time with direct access. Worst-Case: O(log n) when removing from a treeified bucket.



LinkedHashmap – order gets maintained, It is not threadsafe just like HashMap

LinkedHashmap has a double linked list that store all the entries inside it in the order in which you inserted. It is slower and takes more memory in comparison to hashmap.

Its constructor can take three values – initialCapacity, loadFactor and accessOrder

accessOrder is by default false. (which means storing as per insertion order, true for storing as per access order)

```
LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>( initialCapacity: 11, loadFactor: 0.3f, accessOrder: false);
```

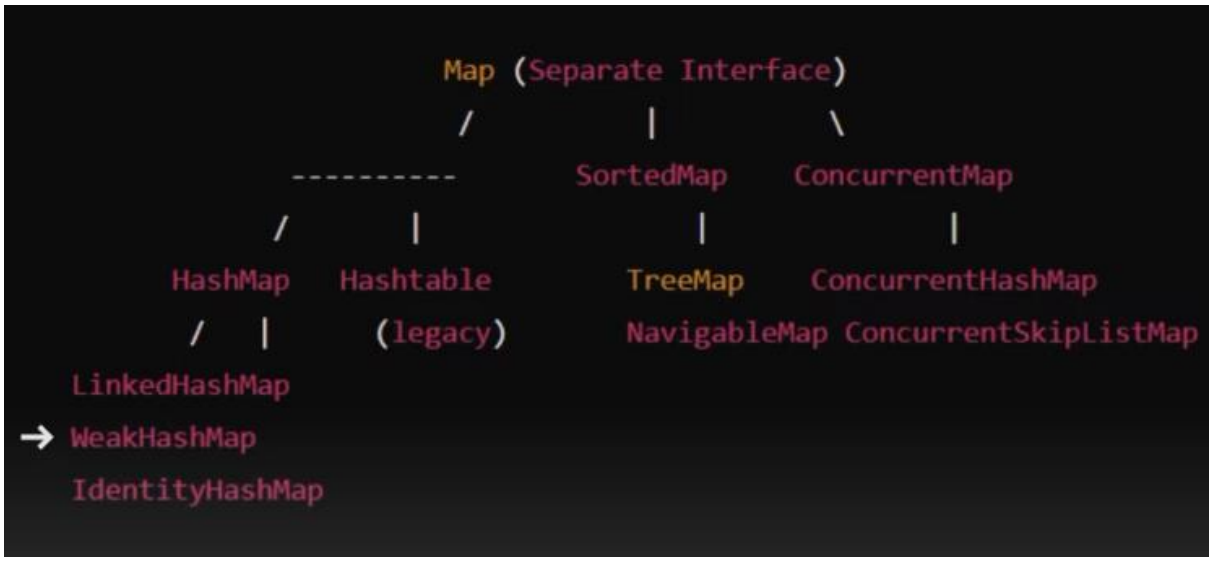
```java
LinkedHashMap<String, Integer> linkedHashMap = new LinkedHashMap<>( initialCapacity: 11, loadFactor: 0.3f, accessOrder: true);
linkedHashMap.put("Orange", 10);
linkedHashMap.put("Apple", 20);
linkedHashMap.put("Guava", 13);

linkedHashMap.get("Apple");
linkedHashMap.get("Orange");

for(Map.Entry<String, Integer> entry : linkedHashMap.entrySet()){
    System.out.println(entry.getKey() + " : " + entry.getValue());
}
// Guava : 13
// Apple : 20
// Orange : 10
```

When accessOrder is true, it gives behaviour of LRU cache. Least recently used item is on starting and latest used item on end. So we can remove the LRU items from top.

```java
HashMap<String, Integer> hashMap = new HashMap<>();
LinkedHashMap<String, Integer> linkedHashMap1 =  new LinkedHashMap<>(hashMap); // convert HashMap into LinkedHashMap

hashMap.put("Shubham", 91);
hashMap.put("Bob", 80);
hashMap.put("Akshit", 78);

Integer res = hashMap.getOrDefault( key: "Vipul",  defaultValue: 0);
hashMap.putIfAbsent("Shubham", 92);
System.out.println(hashMap);  // {Bob=80, Shubham=91, Akshit=78}
```

```java
public class LRUCache<K, V> extends LinkedHashMap<K, V> {
    private int capacity;

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity;
    }

    public LRUCache(int capacity){
        super(capacity, loadFactor: 0.75f, accessOrder: true);
        this.capacity = capacity;
    }
    public static void main(String[] args) {
        LRUCache<String, Integer> studentMap = new LRUCache<>( capacity: 3);
        studentMap.put("Bob", 99);
        studentMap.put("Alice", 89);
        studentMap.put("Ram", 91);
        studentMap.put("Vipul", 89);

        System.out.println(studentMap.get("Bob")); // null
        System.out.println(studentMap); // {Alice=89, Ram=91, Vipul=89}
    }
}
```

WeakHashMap




```
Phone phone = new Phone( brand: "Apple",  model: "16 pro");  // phone is strong reference
System.out.println(phone);
phone = null;
// After this JVM will see that no reference variable is pointing to this
// memory location new Phone("Apple", "16 pro"), so delete it from heap to make that memory available

System.out.println(phone); // null
System.gc(); // it will suggest JVM to run GC (not recommended)
```

```
Phone phone = new Phone( brand: "Samsung",  model: "S24"); // strong reference, GC won't clear its memory
WeakReference<Phone> phoneWeakReference = new WeakReference<>(new Phone( brand: "Apple",  model: "16 pro"));
// JVM when detects a weak reference then it uses internal algo to determine whether it can be removed, so remove it
// If the GC has not yet collected the object, you can retrieve it using the get() method

System.out.println(phoneWeakReference.get()); // Phone{brand='Apple', model='16 pro'}
System.gc(); // manually suggesting JVM to do GC
try {
    Thread.sleep( millis: 5000);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
System.out.println(phoneWeakReference.get());  // null
```

An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use.

String literals are stored in string pool due to which they are strong referenced throughout the lifecycle of program. That's why if we use key as string literal then it won't be removed after GC run.

Example with non literal string –

```
WeakHashMap<String, Image> imageCache = new WeakHashMap<>();
imageCache.put(new String( original: "img1"), new Image( name: "image 1"));
imageCache.put(new String( original: "img2"), new Image( name: "image 2"));
System.out.println(imageCache); // {img1=Image{name='image 1'}, img2=Image{name='image 2'}}
System.gc();
Thread.sleep( millis: 5000);
System.out.println(imageCache); // {}
```

IdentityHashMap

equals method of the String class in Java is overridden to compare **the content** (logical equality) of two strings rather than their memory addresses.

The overridden hashCode method in the String class computes the hash code based on the content of the string, rather than its memory address.

```
String str1 = "arjun";
String str2 = "arjun";
String str3 = new String( original: "arjun");

// Using equals() to compare content
System.out.println(str1.equals(str2)); // true, content is the same
System.out.println(str1.equals(str3)); // true, content is the same

// Using == to compare references
System.out.println(str1 == str2); // true, both refer to the same object in the string pool
System.out.println(str1 == str3); // false, str3 is a new object in heap memory
```

```
String key1 = new String( original: "key");
String key2 = new String( original: "key");
Map<String, Integer> map = new HashMap<>();
map.put(key1, 1); // hashcode1
map.put(key2, 2); // hashcode1 ----> replace
System.out.println(map); //{key=2}
map.put("key", 3); // hashcode1 ----> replace
System.out.println(map); // {key=3}
```

Whether in ur class u override the hashcode or not. In identityHashMap only the hashcode of Object class will be used which generate hashcode based on memory address. And it won't use equals() method to compare but it uses == operator which will check memory address.

```
String key1 = new String( original: "key");
String key2 = new String( original: "key");
System.out.println(System.identityHashCode(key1)); // 1705736037
System.out.println(System.identityHashCode(key2)); // 455659002
System.out.println(key1.hashCode()); // 106079
System.out.println(key2.hashCode()); // 106079

Map<String, Integer> map = new IdentityHashMap<>();
// uses Identify hashcode and ==

map.put(key1, 1); // hashcode1
map.put(key2, 2); // hashcode2
System.out.println(map); // {key=1, key=2}
map.put("key", 3); // hashcode3
map.put("key", 4); // hashcode3
System.out.println(map); // {key=1, key=4, key=2}
System.out.println(map.get("key")); // 4
System.out.println(map.get(key1)); // 1
```

Comparable – used for sorting of class

Comparator is used when we need to give custom logic.

list.sort(null) means we want natural ordering. For that class needs to implement Comparable<T> interface and implements compareTo() method and write sorting logic inside it. U can sort only based on 1 criteria. Using Comparable we set natural ordering mechanism for classes.

```
public interface Comparable<T> {
    /** Compares this object with the specified object for order.  Returns a ...*/
    @Contract(pure = true)
    public int compareTo( @NotNull T o);
}
```

```
class Student implements Comparable<Student>{
    private String name;
    private double gpa;
    public Student(String name, double gpa) {
        this.name = name;
        this.gpa = gpa;
    }
    public String getName() {return name;}
    public void setName(String name) {this.name = name;}
    public double getGpa() {return gpa;}
    public void setGpa(double gpa) {this.gpa = gpa;}

    @Override
    public String toString() {
        return "Student{" +
                "name='" + name + '\'' +
                ", gpa=" + gpa +
                '}';
    }

    // 4.compareTo(3)    ---> 4, 3
    @Override
    public int compareTo(Student o) {
        // we want high gpa student to come first
        // if(this.getGpa() > o.getGpa()) return -1;
        // else if(this.getGpa() < o.getGpa()) return 1;
        // else return 0;

        return Double.compare(o.getGpa(), this.getGpa());
    }
}
```

```java
List<Integer> numbers = new ArrayList<>();
numbers.add(2); numbers.add(20); numbers.add(1);
numbers.sort( c: null); // natural order
System.out.println(numbers); // [1, 2, 20]

List<Student> list = new ArrayList<>();
list.add(new Student( name: "Charlie",  gpa: 3.4));
list.add(new Student( name: "Arjun",  gpa: 3.7));
list.add(new Student( name: "Ishan",  gpa: 3.3));
list.add(new Student( name: "Bob",  gpa: 3.9));
list.sort( c: null);
System.out.println(list);
```

O/P – descending order of gpa

SortedMap – its an interface and TreeMap is its implementation

SortedMap is an interface that extends Map and guarantees that the entries are sorted based on the keys, either in their natural ordering or by a specified Comparator.

Null key is not allowed as sorting can't happen, it will throw NPE

TreeMap internal implementation is Red Black tree which is self balancing binary search tree.

Time complexity = O(logn)

```java
SortedMap<Integer, String> map = new TreeMap<>();
map.put(91, "Vivek");
map.put(99, "Shubham");
map.put(78, "Mohit");
map.put(77, "Vipul");
map.get(77);
map.containsKey(78);
map.containsValue("Vivek");
System.out.println(map); // {77=Vipul, 78=Mohit, 91=Vivek, 99=Shubham}
System.out.println(map.firstKey()); // 77
System.out.println(map.lastKey()); // 99
System.out.println(map.headMap( toKey: 91)); // {77=Vipul, 78=Mohit}
System.out.println(map.tailMap( fromKey: 91)); // {91=Vivek, 99=Shubham}

// for descending order => pass comparator
SortedMap<Integer, String> map1 = new TreeMap<>((a, b)-> b-a);
```

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
```

```
public interface NavigableMap<K,V> extends SortedMap<K,V> {
```

NavigableMap extends SortedMap, providing more powerful navigation options such as finding the closest matching key or retrieving the map in reverse order.

```
NavigableMap<Integer, String> navigableMap = new TreeMap<>();
navigableMap.put(1, "One");
navigableMap.put(2, "Five");
navigableMap.put(3, "Three");
System.out.println(navigableMap.lowerKey(4));   // <  // 3
System.out.println(navigableMap.ceilingKey(3)); // >= // 3
System.out.println(navigableMap.higherKey(3));  //  > // null
System.out.println(navigableMap.descendingMap()); //{3=Three, 2=Five, 1=One}
```

Hashtable

- All methods are synchronized (get, put, etc..)
- No null key or value
- Replaced by ConcurrentHashMap
- Slower than HashMap
- Implements Map
- Only LinkedList used in case of collision no RBT

```
Hashtable<Integer, String> hashtable = new Hashtable<>();
hashtable.put(1, "Apple");
hashtable.put(2, "Banana");
hashtable.put(3, "Cherry");
System.out.println(hashtable);
hashtable.get(2);
hashtable.containsKey(3);
hashtable.remove( key: 1);
```

```
Hashtable<Integer, String> map = new Hashtable<>();
Thread t1 = new Thread(()->{
    for(int i=0;i<1000;i++){
        map.put(i, "t1");
    }
});
Thread t2 = new Thread(()->{
    for(int i=1000;i<2000;i++){
        map.put(i, "t2");
    }
});
t1.start(); t2.start();
try {
    t1.join(); t2.join();
}catch (Exception e){}

System.out.println(map.size()); // 2000 with Hashtable but <2000 with HashMap
```

ConcurrentHashMap – It is more efficient than HashTable

```
ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
// Java 7 --> segment based locking --> CHM divided into 16 segments --> smaller hashmaps
// each segments has its independent locking
// only the segment being written to or read from is locked
// read : doesn't require locking unlesss there is a write operation happening on the same segment
// write : lock

// java 8 ---> no segmentation
//        ---> Compare And Swap approach --> np locking except resizing or collision
// Thread A last saw --> x = 45
// Thread A work --> x to 50
// if x is still 45, then change it to 50 else don't change and retry
// If CAS failing multiple times, then thread will wait for short random time and then retry
// put --> index
```

ImmutableMap – cannot change the content once instantiated

```
Map<String, Integer> map1 = new HashMap<>();
map1.put("A", 1);
map1.put("B", 2);
Map<String, Integer> map2 = Collections.unmodifiableMap(map1); // returns unmodifiable view
// map2.put("C", 3);       // throws exception

// after java 9 - other ways of creating immutable map
Map<String, Integer> map3 = Map.of( k1: "Shubham", v1: 98, k2: "Arjun", v2: 78);
// Map.of() is limited to only 1 key-value entries
System.out.println(map3); // {Arjun=78, Shubham=98}

// no limitation on no of entries
Map.ofEntries(Map.entry( k: "Akshit", v: 89), Map.entry( k: "Arjun", v: 89));
```

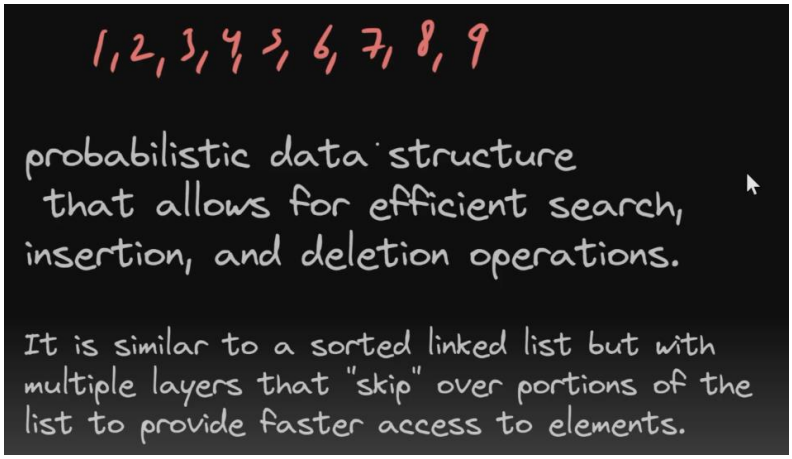
EnumMap –

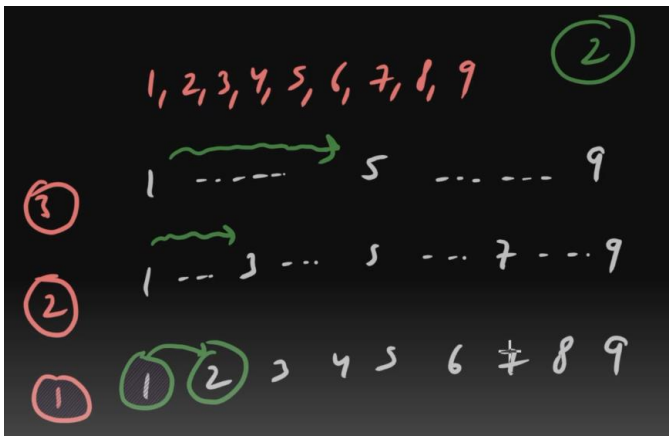
- directly implements map interface
- knows all possible keys in advance , is more efficient compared to HashMap
- No hashing needed, values directly mapped to ordinal/index
- Maintains order of the enum constants' ordinal values

```java
enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
public class Test {
    public static void main(String[] args) {
        // array of size same as enum
        Map<Day, String> map = new EnumMap<>(Day.class);
        map.put(Day.TUESDAY, "walk");
        map.put(Day.MONDAY, "gym");
        System.out.println(Day.MONDAY.ordinal()); // 0
        System.out.println(map);  // {MONDAY=gym, TUESDAY=walk}
```

ConcurrentSkipListMap –

- thread safe and sorted in order of its keys.
- data entry stored in skiplist data structure

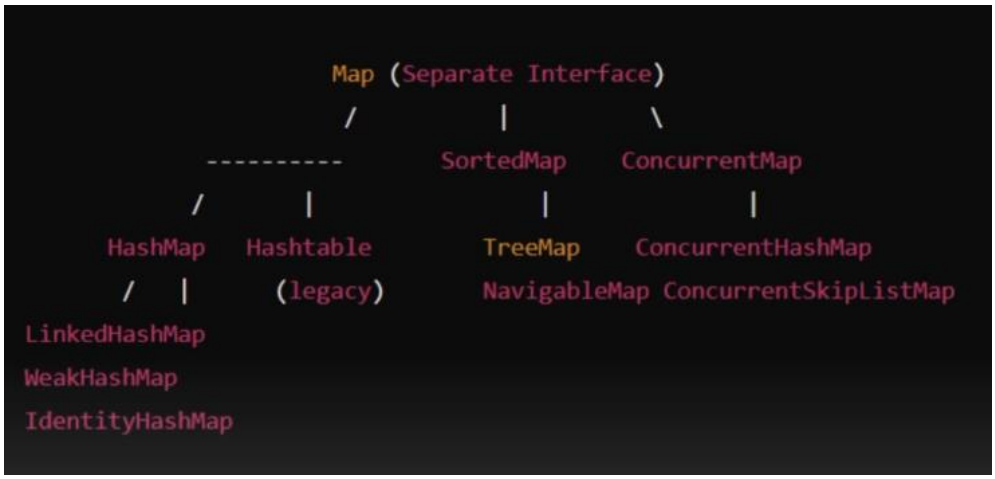




```java
ConcurrentSkipListMap<String, Integer> map = new ConcurrentSkipListMap<>();
```

```java
public class ConcurrentSkipListMap<K,V> extends AbstractMap<K,V>
    implements ConcurrentNavigableMap<K,V>, Cloneable, Serializable {
```

```java
public interface ConcurrentNavigableMap<K,V>
    extends ConcurrentMap<K,V>, NavigableMap<K,V>
```



Streams

- Java 8 -> minimal code, functional programming (treating function as variable)
- new features -> lambda expression, Streams, Date & Time API

```java
class Task implements  Runnable{
    @Override
    public void run() {
        System.out.println("Hello");
    }
}
public class Test {
    public static void main(String[] args) {
        // Lambda expression
        // lambda expression is an anonymous function ( no name, no return type, no access modifier )
        // Used to implement FunctionalInterface(interfaces having only 1 public abstract method)

        Thread t1 = new Thread(new Task());
        Thread t2 = new Thread(()-> System.out.println("Hello"));
```

```java
@FunctionalInterface
interface MathOperation{
    int operate(int a, int b);
}
public class Test {
    public static void main(String[] args) {
        MathOperation sum = (a, b) -> a+b; // using interface reference to hold lambda expression
        MathOperation subtract = (a, b) -> a-b;
        System.out.println(sum.operate( a: 2,  b: 3)); // 5
```

Predicate → functional interface (boolean valued function)

```java
@FunctionalInterface
public interface Predicate<T> {

    Evaluates this predicate on the given argument.
    Params: t – the input argument
    Returns: true if the input argument matches the predicate, otherwise false
    boolean test(T t);
```

```java
Predicate<Integer> isEven = x -> x%2==0;
System.out.println(isEven.test( t: 10)); // true

Predicate<String> isWordStartsWithA = x -> x.toLowerCase().startsWith("a");
Predicate<String> isWordEndsWithT = x -> x.toLowerCase().endsWith("t");
Predicate<String> and = isWordStartsWithA.and(isWordEndsWithT);
System.out.println(and.test( t: "Ankit")); // true
```

Function → work for you

```java
@FunctionalInterface
public interface Function<T, R> {

    Applies this function to the given argument.
    Params: t – the function argument
    Returns: the function result
    R apply(T t);
```

```java
Function<Integer, Integer> doubleIt = x -> 2*x;
Function<Integer, Integer> tripleIt = x -> 3*x;
System.out.println(doubleIt.apply( t: 100)); // 200
System.out.println(doubleIt.andThen(tripleIt).apply( t: 20)); // 120
System.out.println(doubleIt.compose(tripleIt).apply( t: 20)); // 200 , it first call tripleIt then doubleIt

Function<Integer, Integer> identity = Function.identity();
System.out.println(identity.apply( t: 5)); // 5
```

Consumer →

```java
@FunctionalInterface
public interface Consumer<T> {

    Performs this operation on the given argument.
    Params: t – the input argument
    void accept(T t);
```

```java
Consumer<Integer> consumer = n -> System.out.println(n);
consumer.accept(5); // 5
List<Integer> list = Arrays.asList(1, 2, 3);
Consumer<List<Integer>> printList = x ->{
    for(Integer i: x){
        System.out.println(i);
    }
};
printList.accept(list); // 1, 2, 3
```

Supplier →

```java
@FunctionalInterface
public interface Supplier<T> {

    Gets a result.
    Returns: a result

    T get();
}
```

```java
Supplier<String> giveHelloWorld = () -> "Hello World";
System.out.println(giveHelloWorld.get()); // Hello World
```

```java
Predicate<Integer> predicate = x -> x%2==0;
Function<Integer, Integer> function = x -> x*x;
Consumer<Integer> consumer = x -> System.out.println(x);
Supplier<Integer> supplier = () -> 100;

if(predicate.test(supplier.get())){
    consumer.accept(function.apply(supplier.get())); // 10000
}

// BiPredicate, BiConsumer, BiFunction
BiPredicate<Integer, Integer> isSumEven = (x, y) -> (x+y)%2==0;
System.out.println(isSumEven.test(5, 5)); // true

BiConsumer<Integer, Integer> biConsumer = (x, y) ->{
    System.out.println(x);
    System.out.println(y);
};

BiFunction<String ,String ,Integer> biFunction = (x, y) -> (x+y).length();
System.out.println(biFunction.apply("hello ", "ji")); // 8
```

UnaryOperator →

```java
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T> {
```

BinaryOperator →

```java
@FunctionalInterface
public interface BinaryOperator<T> extends BiFunction<T,T,T> {
```

```
UnaryOperator<Integer> a = x -> 2*x;
BinaryOperator<Integer> b = (x, y) -> x+y;
```

Method Reference → use method without invoking in place of lambda expression

```
    public static void main(String[] args) {
        List<String> students = Arrays.asList("Ram", "Shyam", "Ghanshyam");
        students.forEach(x -> System.out.println(x));
        students.forEach(System.out::println);

        // Constructor Reference
        List<String> names = Arrays.asList("A", "B", "C");
        List<MobilePhone> mobilePhoneList1 = names.stream().map(x -> new MobilePhone(x)).collect(Collectors.toList());
        List<MobilePhone> mobilePhoneList2 = names.stream().map(MobilePhone::new).collect(Collectors.toList());
    }
}
class MobilePhone{
    private String name;
    public MobilePhone(String name) { this.name = name; }
}
```

Streams

- process collections of data in a functional and declarative manner
- Simplify Data Processing
- Improve Readability and Maintainability
- Enable Easy Parallelism

- How to Use Streams ?
  Source, intermediate operations & terminal operation

```
List<Integer> numbers = Arrays.asList(1,2,3,4,5,6);
// finding total even numbers
System.out.println(numbers.stream().filter(x->x%2==0).count()); // 3

// Creating Streams
// 1. From collections
List<Integer> list = Arrays.asList(1,2,3,4,5,6);
Stream<Integer> stream = list.stream();
// 2. From Arrays
String[] array = {"a", "b", "c", "d"};
Stream<String> stream1 = Arrays.stream(array);
// 3. Using Stream.of()
Stream<String> stream2 = Stream.of( ...values: "a", "b");
// 4. Infinite streams
Stream<Integer> generate = Stream.generate(() -> 1); // 1,1,1,1,1,1,........
Stream<Integer> iterate  = Stream.iterate(1, x -> x + 1);// 1,2,3,4,5,6.........
```

```java
// Intermediate operations transform a stream into another stream
// They are lazy, meaning they dont execute until a terminal operation is invoked

// 1. filter
List<String> list = Arrays.asList("Akshit", "Ram", "Shyam", "Ghanshyam", "Akshit");
Stream<String> filteredStream = list.stream().filter(x -> x.startsWith("A"));
// no filtering at this point

long res = list.stream().filter(x -> x.startsWith("A")).count();
System.out.println(res); // 2

// 2. map
Stream<String> stringStream = list.stream().map(x -> x.toUpperCase());
Stream<String> stringStream2 = list.stream().map(String::toUpperCase);

// 3. sorted
Stream<String> sorted = list.stream().sorted();
Stream<String> sorted2 = list.stream().sorted((a, b) -> a.length() - b.length());

// 4. distinct
Stream<String> distinct = list.stream().filter(x -> x.startsWith("A")).distinct();

// 5. limit
Stream<Integer> limit = Stream.iterate( seed: 1, x -> x + 1).limit( maxSize: 10);

// 6. skip
Stream<Integer> limit1 = Stream.iterate( seed: 1, x -> x + 1).skip( n: 10).limit( maxSize: 100); // skip starting 10 values
```

```java
// 7. peek
// performs an action on each element as it is consumed
long n = Stream.iterate( seed: 1, x->x+1).skip( n: 10).limit( maxSize: 20).peek(x-> System.out.println(x)).count();
System.out.println(n); // 20
```

flatMap() is used when you have a **stream of collections (or arrays), and you want to "flatten" them into a single stream of their individual elements.**

```java
// 8. flatMap
// Handle streams of collections, lists, or arrays where each element is itself a collection
// Flatten nested structures (e.g., lists within lists) so that they can be processed as a single sequence of elements
// Transform and flatten elements at the same time.
List<List<String>> list = Arrays.asList(
        Arrays.asList("apple", "banana"),
        Arrays.asList("orange", "kiwi"),
        Arrays.asList("pear", "grape")
);
List<String> list1 = list.stream().flatMap(x -> x.stream()).map(String::toUpperCase).toList();
System.out.println(list1); // [APPLE, BANANA, ORANGE, KIWI, PEAR, GRAPE]

List<String> sentences = Arrays.asList(
        "Hello world",
        "Java streams are powerful",
        "flatMap is useful"
);
System.out.println(sentences.stream().flatMap(sentence -> Arrays.stream(sentence.split( regex: " ")))
                                     .map(String::toUpperCase).toList());
// [HELLO, WORLD, JAVA, STREAMS, ARE, POWERFUL, FLATMAP, IS, USEFUL]
```

Terminal operators –

```java
// 1. collect
list.stream().skip( n: 1).collect(Collectors.toList());
list.stream().skip( n: 1).toList(); // need jdk >= 16

// 2. forEach
list.stream().forEach(x -> System.out.println(x));

// 3. reduce - repeatedly combines elements of the stream using this BiOperator
Optional<Integer> optionalInteger = list.stream().reduce((x, y) -> x + y);
System.out.println(optionalInteger.get());

// 4. count

// 5. anyMatch, allMatch, noneMatch
boolean b = list.stream().anyMatch(x -> x % 2 == 0);  // true
boolean b1 = list.stream().allMatch(x -> x > 0); // true
boolean b2 = list.stream().noneMatch(x -> x < 0); // true

// 6. findFirst, findAny
System.out.println(list.stream().findFirst().get()); // 1
System.out.println(list.stream().findAny().get()); // any no from list
```

```java
// 7. toArray()
Object[] array = Stream.of( ...values: 1, 2, 3, 4, 5).toArray();

// 8. min/max
System.out.println(Stream.of( ...values: 1,2,3,4,5).max((a, b) -> b-a).get()); // 1 , sort descending and give last
System.out.println(Stream.of( ...values: 1,2,3,4,5).min(Comparator.naturalOrder()).get()); // 1
```

```java
// 9. forEachOrdered
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
list.parallelStream().forEach(System.out::println); // some random order [3,5,4,2,1]

list.parallelStream().forEachOrdered(System.out::println); // [1,2,3,4,5]
// things are going parallelly but order is also maintained
```

ParallelStream allows us to process elements of a stream in **parallel**, leveraging multiple CPU cores.

```java
// Example
// Streams cannot be reused after a terminal operation has been called
List<String> names = Arrays.asList("Anna", "Bob", "Charlie", "David");
Stream<String> stream = names.stream();
stream.forEach(System.out::println);
List<String> list = stream.map(String::toUpperCase).toList(); // IllegalStateException
```

```java
// Example : Filtering and Collecting Names
List<String> list = Arrays.asList("Anna", "Bob", "charlie", "david");
System.out.println(list.stream().filter(x -> x.length() > 3 ).collect(Collectors.toList()));
// [Anna, charlie, david]

// Example : Squaring and Sorting Numbers
List<Integer> list1 = Arrays.asList(5, 3, 2, 7, 1);
System.out.println(list1.stream().map(x->x*x).sorted().collect(Collectors.toList()));
// [1, 4, 9, 25, 49]

// Example : Summing Values
List<Integer> list2 = Arrays.asList(1, 2, 3, 4, 5, 6);
System.out.println(list2.stream().reduce((x, y) -> x + y).get()); // 21

// Example : Counting occurence of a Character
String s = "Hello World";
IntStream chars = s.chars();
System.out.println(s.chars().filter(c -> c=='l').count()); // 3
```

```java
List<String> list = Arrays.asList("Alice", "Bob", "Charlie", "David");
Stream<String> stream = list.stream().filter(name -> {
    System.out.println("Filtering : " + name);
    return name.length() > 3;
});

System.out.println("Before Terminal Operation");

List<String> strings = stream.collect(Collectors.toList());
System.out.println("After terminal operation");

// Before Terminal Operation
// Filtering : Alice
// Filtering : Bob
// Filtering : Charlie
// Filtering : David
// After terminal operation
```

```java
public static void main(String[] args) {
    // A type of stream that enables parallel processing of elements
    // Allowing multiple threads to process parts of the stream simultaneously
    // This can significantly improve performance for large data sets
    // workload is distributed across multiple threads
    long startTime = System.currentTimeMillis();
    List<Integer> list = Stream.iterate( seed: 1, x -> x + 1).limit( maxSize: 20000).toList();
    List<Long> factorialsList = list.stream().map(x -> factorial(x)).toList();
    long endTime = System.currentTimeMillis();
    System.out.println("Time taken with sequential stream: " + (endTime - startTime) + "ms"); // 235ms

    startTime = System.currentTimeMillis();
    factorialsList = list.parallelStream().map(x -> factorial(x)).toList();
    endTime = System.currentTimeMillis();
    System.out.println("Time taken with parallel stream : " + (endTime - startTime) + "ms"); // 47ms

    // Parallel streams are most effective for CPU-intensive or large datasets where tasks are independent
    // They may add overhead for simple tasks or small datasets
}
private static long factorial(int n) {...}
```

We can convert parallelStream to sequential –

```
List<Integer> list = Stream.iterate( seed: 1, x -> x + 1).limit( maxSize: 20000).toList();
Stream<Long> sequential = list.parallelStream().map(x -> factorial(x)).sequential();
```

parallelStream don't work here -

```
// Cumulative Sum
// [1,2,3,4,5] --> [1,3,6,10,15]
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
AtomicInteger sum = new AtomicInteger( initialValue: 0);
List<Integer> cumulativeSum = list.parallelStream().map(x -> sum.addAndGet(x)).toList();
System.out.println("Expected : [1,3,6,10,15]");
System.out.println("Actual : " + cumulativeSum); // [13, 15, 12, 9, 5] which is Wrong
```

Collectors – It is a utility class which provides a set of method to create a common collectors

```
// 1. Collecting to a list
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
List<String> res = names.stream()
        .filter(name -> name.startsWith("A"))
        .collect(Collectors.toList());
System.out.println(res); // [Alice]

// 2. Collecting to a Set
List<Integer> nums = Arrays.asList(1,2,2,3,3,5,6,66,66);
Set<Integer> set = nums.stream().collect(Collectors.toSet());
System.out.println(set); // [1, 2, 66, 3, 5, 6]

// 3. Collecting to s specific collection
ArrayDeque<String> collect = names.stream().collect(Collectors.toCollection(() -> new ArrayDeque<>()));
System.out.println(collect); // [Alice, Bob, Charlie]
```

```
// 4. Joining Strings - Concatenates stream elements into a single string
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
String concatenatedNames = names.stream().map(String::toUpperCase).collect(Collectors.joining( delimiter: ", "));
System.out.println(concatenatedNames); // ALICE, BOB, CHARLIE

// 5. Summarizing Data - generates statistical summary (count, min, sum, average, max)
List<Integer> numbers = Arrays.asList(1,2,3,4,5);
IntSummaryStatistics stats = numbers.stream().collect(Collectors.summarizingInt(x -> x));
System.out.println(stats.getCount()); // 5
System.out.println(stats.getSum()); // 15
System.out.println(stats.getMax()); // 5
System.out.println(stats.getMin()); // 1
System.out.println(stats.getAverage()); // 3.0

// 6. Calculating Averages
Double average = numbers.stream().collect(Collectors.averagingInt(x -> x));
System.out.println(average); // 3.0

// 7. Counting Elements
Long count = numbers.stream().collect(Collectors.counting());
System.out.println(count); // 5
```

```java
// 8. Grouping Elements
List<String> words = Arrays.asList("hello", "world", "java", "streams", "collecting");
// suppose we want to group strings based on length
Map<Integer, List<String>> map = words.stream().collect(Collectors.groupingBy(x -> x.length()));
System.out.println(map); //{4=[java], 5=[hello, world], 7=[streams], 10=[collecting]}

Map<Integer, String> map2 = words.stream().collect(Collectors.groupingBy(x -> x.length(),
        Collectors.joining( delimiter: "-")));   // perform this on all groups
System.out.println(map2); // {4=java, 5=hello-world, 7=streams, 10=collecting}

Map<Integer, Long> map3 = words.stream().collect(Collectors.groupingBy(x -> x.length(),
        Collectors.counting()));
System.out.println(map3); // {4=1, 5=2, 7=1, 10=1}

TreeMap<Integer, Long> map4 = words.stream().collect(Collectors.groupingBy(x -> x.length(),
        () -> new TreeMap<>(), Collectors.counting()));
System.out.println(map4); // {4=1, 5=2, 7=1, 10=1}
```

```java
// 9. Partitioning Elements - partition elements into 2 groups (true or false) based on a predicate
List<String> words = Arrays.asList("hello", "world", "java", "streams", "collecting");
Map<Boolean, List<String>> map = words.stream().collect(Collectors.partitioningBy(x -> x.length() > 4));
System.out.println(map); // {false=[java], true=[hello, world, streams, collecting]}

// 10. Mapping and Collecting - applies a mapping function before collecting
System.out.println(words.stream().collect(Collectors.mapping(x -> x.toUpperCase(), Collectors.toList())));
// [HELLO, WORLD, JAVA, STREAMS, COLLECTING]

// 11. collectingAndThen - first collect elements using a collector & then apply a finishing function to the result
List<String> collect = words.stream().collect(Collectors.collectingAndThen(
        Collectors.toList(), list -> Collections.unmodifiableList(list)
));
```

```java
// Example 1: Collecting Names by Length
List<String> l1 = Arrays.asList("Anna", "Bob", "Alexander", "Brian", "Alice");
System.out.println(l1.stream().collect(Collectors.groupingBy(x->x.length())));

// Example 2: Counting Word Occurrences
String s = "hello world hello java world";
System.out.println(Arrays.stream(s.split( regex: " ")).collect(Collectors.groupingBy(x -> x, Collectors.counting())));
// {java=1, world=2, hello=2}

// 3. Partitioning Even and Odd Number
List<Integer> l2 = Arrays.asList(1,2,3,4,5,6,7);
System.out.println(l2.stream().collect(Collectors.partitioningBy(x -> x%2==0)));
// {false=[1, 3, 5, 7], true=[2, 4, 6]}

// 4. Summing Values in a Map
Map<String, Integer> items = new HashMap<>();
items.put("Apple", 20);
items.put("Banana", 25);
items.put("Orange", 15);
System.out.println(items.values().stream().reduce((x, y) -> x+y).get()); // 60
System.out.println(items.values().stream().collect(Collectors.summingInt(x->x))); // 60
```

```java
// 5. Creating a Map from Stream Elements of Words and their length
List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry");
                                                // in toMap we pass KeyMapper & ValueMapper
System.out.println(fruits.stream().collect(Collectors.toMap(x->x.toUpperCase(), x->x.length())));
// {CHERRY=6, APPLE=5, BANANA=6}

// 6. Count occurrence of each word using toMap with mergeFunction
List<String> words2 = Arrays.asList("apple", "banana", "apple", "orange", "banana", "apple");
System.out.println(words2.stream().collect(Collectors.toMap(k->k, v->1, (x, y)->x+y)));
// {orange=1, banana=2, apple=3}
```

Primitive Streams

```java
Integer[] nums1 = {1,2,3,4,5};
Stream<Integer> stream1 = Arrays.stream(nums1);

int[] nums = {1,2,3,4,5};
IntStream stream = Arrays.stream(nums);

IntStream range = IntStream.range(1, 5);  // use boxed() to convert to wrapper class stream
System.out.println(range.boxed().collect(Collectors.toList())); // [1, 2, 3, 4]

System.out.println(IntStream.rangeClosed(1,5).boxed().collect(Collectors.toList())); // [1,2,3,4,5]

IntStream intStream = IntStream.of( ...values: 10, 342, 5, 6);

IntStream ints = new Random().ints( streamSize: 5);
System.out.println(ints.boxed().collect(Collectors.toList()));
// [-321177229, 1451905699, 912836823, -947953148, 1289541290]

DoubleStream doubles = new Random().doubles( streamSize: 5);
System.out.println(doubles.sum()); // 2.771484444331391
// doubles.min();
// doubles.max();
// doubles.average();
// doubles.summaryStatistics();
// doubles.mapToInt(x -> (int)(x+1));
```

Iterator – used to traverse over the Collection

we are able to use for each loop in ArrayList because of it implements Iterable interface and provides implementation of iterator().

```java
List<Integer> list = new ArrayList<>();
for(int i:list){
    System.out.println(i);
}
// above code is changed to below during compilation
Iterator<Integer> iterator = list.iterator();
while(iterator.hasNext()){
    System.out.println(iterator.next());
}
```

Iterator provides functionality of removing while iterating –

```java
List<Integer> numbers = new ArrayList<>();
numbers.add(1); numbers.add(2); numbers.add(3); numbers.add(4); numbers.add(5);
for(Integer n : numbers){
    if(n%2 == 0) numbers.remove(n); // java.util.ConcurrentModificationException
}

Iterator<Integer> itr = numbers.iterator();
while(itr.hasNext()){
    Integer n = itr.next();
    if(n%2 == 0){
        itr.remove(); // no exception
    }
}
System.out.println(numbers); // [1,3,5]
```

```java
// ListIterator extends Iterator and has more functionalities
ListIterator<Integer> listIterator = numbers.listIterator();
while(listIterator.hasNext()){
    Integer n = listIterator.next();
    if(n%2==0) listIterator.set(-1);
}
System.out.println(numbers); // [1, -1, 3, -1, 5]

// Now the iterator is at the end, let's move backward
while(listIterator.hasPrevious()){
    System.out.println(listIterator.previous());
}
```

Sets

```java
public interface Set<E> extends Collection<E> {
```

```java
// Set is a collection that cannot contain duplicate elements
// find - O(1), insert - O(1)
// Map --> HashMap, LinkedHashMap, TreeMap, EnumMap
// Set --> HashSet, LinkedHashSet, TreeSet, EnumSet
Set<Integer> set = new HashSet<>();
Map<Integer, String> map = new HashMap<>();
Set<Integer> keys = map.keySet();
set.add(12);
set.add(1);
set.add(1);
set.add(67);
System.out.println(set); // [1, 67, 12]   in unordered way

// ordered
Set<Integer> set2 = new LinkedHashSet<>();
set2.add(45);
set2.add(34);
set2.add(99);
System.out.println(set2); // [45, 34, 99]
```

```java
// sorted manner
Set<Integer> set1 = new TreeSet<>();
NavigableSet<Integer> set = new TreeSet<>();
set.add(12);
set.add(46);
set.add(1);
System.out.println(set.contains(12)); // true
System.out.println(set.remove( o: 46)); // true  ....would return false if no not present
set.clear();
System.out.println(set.isEmpty()); // true
for(int i:set){
    System.out.println(i);
}
```

```java
// unmodifiable
Set<Integer> set = Set.of(1,2,3,40,4,532,32,42,414); // we can add more than 10 values
Set<Integer> set1 = Collections.unmodifiableSet(new HashSet<>());
```

Collections.synchronizedSet() wraps the underlying set (e.g., HashSet) and synchronizes **every method** on the set using a **single lock**. This means that **every method call** (like add(), remove(), contains(), etc.) is synchronized using the same lock object. Even **read operations** (like contains() and size()) are locked. Although reads generally don't modify the set, they still have to acquire the lock.

```java
Set<Integer> set = new HashSet<>();
set.add(12);
set.add(46);
set.add(1);

// for thread safety
Set<Integer> integers = Collections.synchronizedSet(set); // all method will be wrapped in synchronized block
                                    // poor performance not recommended
Set<Integer> syncSet = Collections.synchronizedSet(new TreeSet<>());
synchronized (syncSet){
    for(Integer i : syncSet){
        System.out.println(i);
    }
}

Set<Integer> set1 = new ConcurrentSkipListSet<>(); // recommended
```

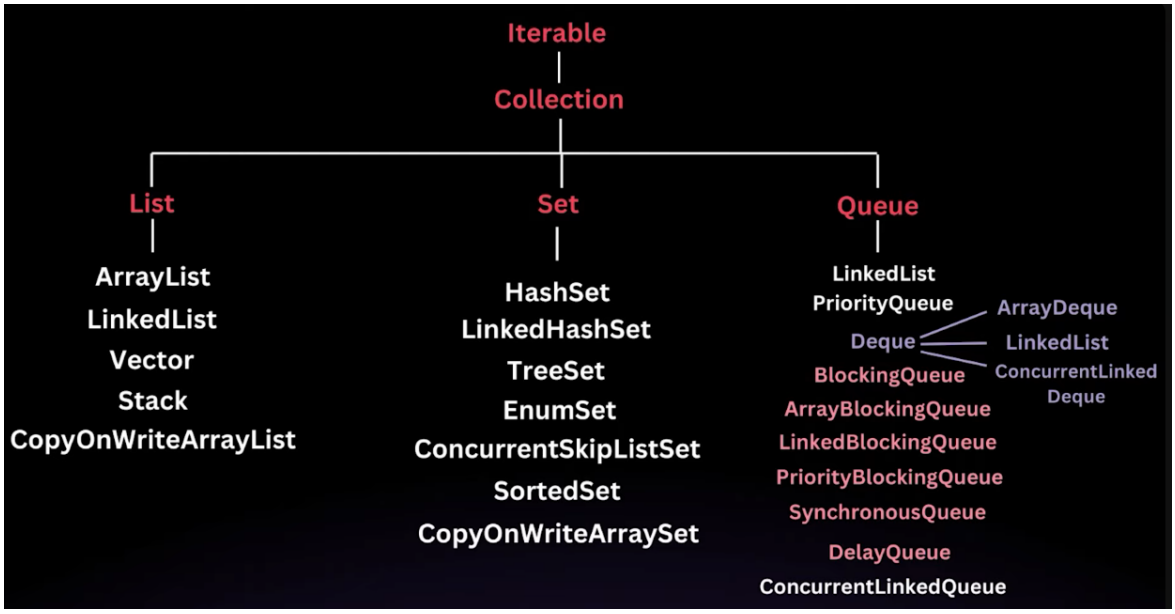
In ConcurrentSkipListSet, data is stored in sorted manner, useful in frequent reads & writes. It is thread safe.

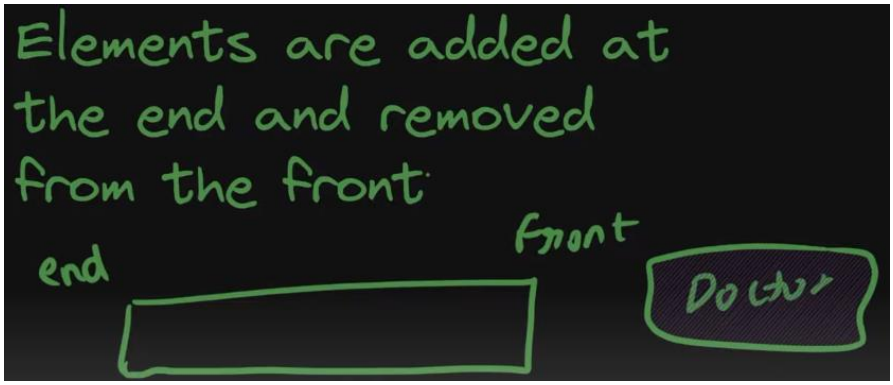
CopyOnWriteArraySet –

- Thread safe
- Copy-On-Write mechanism
- No Sorted
- Iterators Do Not Reflect Modifications
- useful when read operations are frequent

```java
CopyOnWriteArraySet<Integer> copyOnWriteSet = new CopyOnWriteArraySet<>();
ConcurrentSkipListSet<Integer> concurrentSkipListSet = new ConcurrentSkipListSet<>();
for (int i = 1; i <= 5; i++) {
    copyOnWriteSet.add(i);
    concurrentSkipListSet.add(i);
}

// Iterating and modifying CopyOnWriteArraySet
for (Integer num : copyOnWriteSet) {
    System.out.println(num);  // 1 2 3 4 5
    // Attempting to modify the set during iteration
    copyOnWriteSet.add(6);
}
System.out.println(copyOnWriteSet); // 1 2 3 4 5 6

for(Integer num : concurrentSkipListSet){
    System.out.println(num); // 1 2 3 4 5 6
    concurrentSkipListSet.add(6); // it is weakly consistent, it may or may not add 6 to set
}
```

Queues – based on FIFO



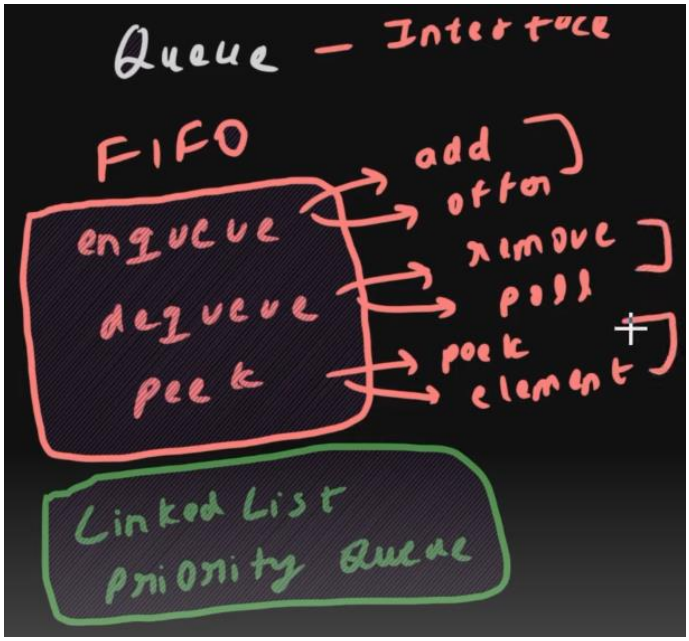



```java
// LinkedList as queue
LinkedList<Integer> list = new LinkedList<>();
list.addLast( e: 1); // enqueue
list.addLast( e: 2);
list.addLast( e: 3);
System.out.println(list); // [1, 2, 3]
Integer i = list.removeFirst(); // dequeue
System.out.println(list); // [2, 3]

Integer f = list.getFirst(); // peek
```

Using Queue interface –

```java
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
```

```java
Queue<Integer> list = new LinkedList<>();
list.add(1); // enqueue
list.add(2);
list.add(3);
System.out.println(list); // [1, 2, 3]
Integer i = list.remove(); // dequeue
System.out.println(list); // [2, 3]

Integer f = list.peek(); // peek
```




```
Queue<Integer> q = new LinkedList<>();
q.add(1);
System.out.println(q.size()); // 1
System.out.println(q.remove()); // 1
System.out.println(q.poll()); // null (better)
System.out.println(q.peek()); // null (better)

System.out.println(q.element()); // throws NoSuchElementException if empty
System.out.println(q.remove()); // throws NoSuchElementException if empty
```

ArrayBlockingQueue is thread safe.

```
Queue<Integer> q = new ArrayBlockingQueue<>( capacity: 2); // fixed size
System.out.println(q.add(1)); // true
System.out.println(q.offer( e: 2)); // true

System.out.println(q.offer( e: 3)); // false ........No Error
q.add(4); // Error : Queue full
```

Priority Queue –

```java
// part of the Queue interface
// orders elements based on their natural ordering (for primitives lowest first)
// custom comparator for customised ordering
// does not allow null elements
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(15);
pq.add(10);
pq.add(30);
pq.add(5);
System.out.println(pq.remove()); // 5
System.out.println(pq.peek()); // 10
System.out.println(pq); // not sorted

while(!pq.isEmpty()){
    System.out.println(pq.poll()); // 10 15 30
}
```

PriorityQueue is implemented as a min-heap by default (for natural ordering)

Min-heap → binary tree in which parent node is smaller than its child node, having smallest element on root

Time Complexity → O(1) in retrieve, O(logn) in insertion/deletion

```java
PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder()); // greatest element on top
PriorityQueue<Integer> pq1 = new PriorityQueue<>((x, y) -> y-x);
```

Deque –

- double-ended queue
- allows insertion and removal of elements from both ends
- versatile than regular queues and stacks because they support all the operations of both

```
INSERTION METHODS
 addFirst(E e): Inserts the specified element at the front.
 addLast(E e): Inserts the specified element at the end.
 offerFirst(E e): Inserts the specified element at the front if possible.
 offerLast(E e): Inserts the specified element at the end if possible.

 REMOVAL METHODS
  removeFirst(): Retrieves and removes the first element.
  removeLast(): Retrieves and removes the last element.
  pollFirst(): Retrieves and removes the first element, or returns null if empty.
  pollLast(): Retrieves and removes the last element, or returns null if empty.

 EXAMINATION METHODS
  getFirst(): Retrieves, but does not remove, the first element.
  getLast(): Retrieves, but does not remove, the last element.
  peekFirst(): Retrieves, but does not remove, the first element, or returns null if empty.
  peekLast(): Retrieves, but does not remove, the last element, or returns null if empty.

 STACK METHODS
  push(E e): Adds an element at the front (equivalent to addFirst(E e)).
  pop(): Removes and returns the first element (equivalent to removeFirst())
```

```java
Deque<Integer> dq = new LinkedList<>(); // useful when insert/delete somewhere in middle

Deque<Integer> deque = new ArrayDeque<>(); // faster iteration, low memory, no null allowed
// ArrayDeque uses circular array, maintaining head and tail reference
// no need to shift elements, just shift head and tail
// size becomes doubled when full
deque.add(10);
deque.add(20);
deque.add(5);
deque.add(25);
System.out.println(deque); // [10, 20, 5, 25]
System.out.println(deque.getFirst()); // 5
System.out.println(deque.getLast()); // 25
deque.removeFirst(); // Removes 5
deque.pollLast(); // Removes 25

for(int n : deque){
    System.out.println(n); // 20 5
}
```

Blocking Queue

- Thread-safe queue
- Wait for queue to become non-empty / wait for space
- Simplify concurrency problems like producer-consumer

- standard queue ---> immediately
  empty --> remove (no waiting)
  full --> add (no waiting)
- Blocking queue
  put --> Blocks if the queue is full until space becomes available
  take --> Blocks if the queue is empty until an element becomes available
  offer --> Waits for space to become available, up to the specified timeout
- Needed when threads need to coordinate with each other

```java
class Producer implements Runnable{
    private BlockingQueue<Integer> queue;
    private int value = 0;
    public Producer(BlockingQueue<Integer> queue){ this.queue = queue; }

    @Override
    public void run() {
        while(true){
            try {
                System.out.println("Producer produced : " + value);
                queue.put(value++);
                Thread.sleep( millis: 1000);
            } catch (InterruptedException e) {...}

        }
    }
}

class Consumer implements Runnable{
    private BlockingQueue<Integer> queue;
    public Consumer(BlockingQueue<Integer> queue){ this.queue = queue; }

    @Override
    public void run() {
        while(true){
            try {
                Integer value = queue.take();
                System.out.println("Consumer consumed : " + value);
                Thread.sleep( millis: 2000);
            } catch (InterruptedException e) {...}

        }
    }
}
```

```java
BlockingQueue<Integer> queue = new ArrayBlockingQueue<>( capacity: 5); // giving capacity is must
// A bounded, blocking queue backed by circular array
// low memory overhead
// uses a single lock for both enqueue and dequeue operations
// more threads --> problem due to frequently blocking
Thread producer = new Thread(new Producer(queue));
Thread consumer = new Thread(new Consumer(queue));
producer.start();
consumer.start();

BlockingQueue<Integer> queue1 = new LinkedBlockingQueue<>(); // default capacity of Integer.MAX_VALUE
// optionally bounded backed by LinkedList
// Uses two separate locks for enqueue and dequeue operations
// Higher concurrency between producers and consumers
// useful when more threads

BlockingQueue<Integer> queue2 = new PriorityBlockingQueue<>(); // default initial capacity (11)
// unbounded --> put wont block as space always available
// Binary Heap as array and can grow dynamically
// Head is based on their natural ordering or a provided Comparator like priority queue

BlockingQueue<String > queue3 = new PriorityBlockingQueue<>( initialCapacity: 11, Comparator.reverseOrder());
queue3.add("apple");
queue3.add("banana");
queue3.add("cherry");
System.out.println(queue3); // [cherry, apple, banana]
```

SynchronousQueue - each insert operation must wait for a corresponding remove operation by another thread and vice versa. It cannot store elements, capacity of at most one element

```java
BlockingQueue<String> queue = new SynchronousQueue<>();
Thread producer = new Thread(()->{
    try {
        System.out.println("Producer is waiting to transfer...");
        queue.put( e: "Hello from producer!");
        System.out.println("Producer has transferred the message");
    }catch (InterruptedException e){
        Thread.currentThread().interrupt();
        System.err.println("Producer was interrupted");
    }
});
Thread consumer = new Thread(()->{
    try {
        System.out.println("Consumer is waiting to receive...");
        String message = queue.take();
        System.out.println("Consumer received : " + message );
    }catch (InterruptedException e){
        Thread.currentThread().interrupt();
        System.err.println("Consumer was interrupted");
    }
});
producer.start();
consumer.start();
```

O/P -

```
Consumer is waiting to receive...
Producer is waiting to transfer...
Producer has transferred the message
Consumer received : Hello from producer!
```

DelayQueue –

Thread-safe unbounded blocking queue
Elements can only be taken from the queue when their delay has expired
Useful for scheduling tasks to be executed after a certain delay
internally priority queue

```
class DelayedTask implements Delayed{
    private final String taskName;
    private final long startTime;

    DelayedTask(String taskName, long delay, TimeUnit unit) {
        this.taskName = taskName;
        this.startTime = System.currentTimeMillis() + unit.toMillis(delay);
    }

    @Override
    public long getDelay(TimeUnit unit) {
        long remaining = startTime - System.currentTimeMillis();
        return unit.convert(remaining, TimeUnit.MILLISECONDS);
    }
    @Override
    public int compareTo(Delayed o) {
        if (this.startTime < ((DelayedTask) o).startTime) { return -1; }
        if (this.startTime > ((DelayedTask) o).startTime) { return 1; }
        return 0;
    }
    public String getTaskName() {
        return taskName;
    }
}
public class Main {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<DelayedTask> delayQueue = new DelayQueue<>();
        delayQueue.put(new DelayedTask( taskName: "Task1",  delay: 5, TimeUnit.SECONDS));
        delayQueue.put(new DelayedTask( taskName: "Task2",  delay: 3, TimeUnit.SECONDS));
        delayQueue.put(new DelayedTask( taskName: "Task3",  delay: 10, TimeUnit.SECONDS));
        while(!delayQueue.isEmpty()){
            DelayedTask task = delayQueue.take();
            System.out.println("Executed : " + task.getTaskName() + " at " + System.currentTimeMillis());
        }
```

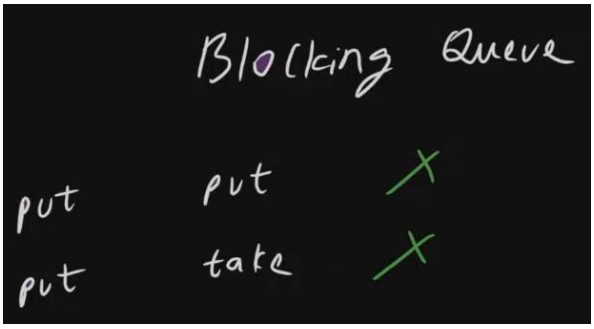
O/P –

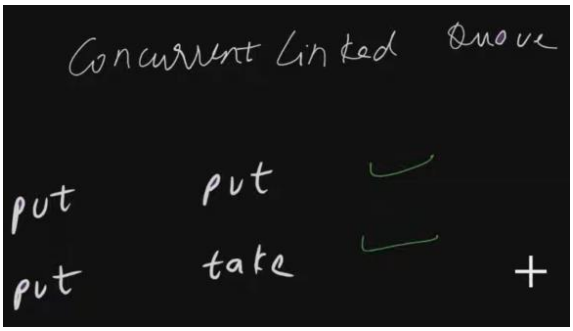
```
Executed : Task2 at 1745032480257
Executed : Task1 at 1745032482250
Executed : Task3 at 1745032487253
```

ConcurrentLinkedQueue –

Used when we don't want to block threads but want to access queue. It supports lock free, thread safe operations.

Use compare & swap method internally like ConcurrentHashMap.






```java
public class TaskSubmissionSystem {
    private static ConcurrentLinkedQueue<String> taskQueue = new ConcurrentLinkedQueue<>();
    public static void main(String[] args) {
        Thread producer = new Thread(()->{
            while(true){
                try{
                    taskQueue.add("Task " + System.currentTimeMillis());
                }catch (Exception e){
                    e.printStackTrace();
                }
            }
        });
        Thread consumer = new Thread(()->{
            while(true){
                try{
                    String task = taskQueue.poll();
                    System.out.println("Processing: " + task);
                }catch (Exception e){
                    e.printStackTrace();
                }
            }
        });
        producer.start();
        consumer.start();
    }
```

ConcurrentLinkedDeque –

```java
// non-blocking, thread-safe double-ended queue
// Compare & Swamp method used
ConcurrentLinkedDeque<String> deque = new ConcurrentLinkedDeque<>();
deque.add("Element1");
deque.addFirst( e: "Element0");
deque.addLast( e: "Element2");
System.out.println(deque); // [Element0, Element1, Element2]

String first = deque.removeFirst();
String last = deque.removeLast();
```