HashMap in Java

In Java, there are **multiple ways to create and initialize a `HashMap`**. Below, I'll explain the different methods, along with examples for each.

---

1. Default Constructor

Creates an empty `HashMap` with the **default initial capacity (16)** and **load factor (0.75)**.

Syntax

```
HashMap<K, V> map = new HashMap<>();
```

```
HashMap<String, Integer> map = new HashMap<>();
map.put("Alice", 25);
map.put("Bob", 30);
```

2. Constructor with Initial Capacity

Creates an empty `HashMap` with a **specified initial capacity** and the **default load factor (0.75)**.

```
HashMap<K, V> map = new HashMap<>(initialCapacity);
```

```
HashMap<String, Integer> map = new HashMap<>(32); // Initial capacity of 32
map.put("Alice", 25);
map.put("Bob", 30);
```

3. Constructor with Initial Capacity and Load Factor

Creates an empty `HashMap` with a **specified initial capacity** and a **custom load factor**.

Syntax

```
HashMap<String, Integer> map = new HashMap<>(32, 0.8f); // Capacity: 32, Load
Factor: 0.8
map.put("Alice", 25);
```

```java
map.put("Bob", 30);
```

4. Constructor with an Existing Map

Creates a `HashMap` and initializes it with the key-value pairs from an **existing map**.

Syntax

```java
Map<String, Integer> existingMap = new HashMap<>();
existingMap.put("Alice", 25);
existingMap.put("Bob", 30);

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

5. Using `Map.of()` (Java 9+)

Creates an **immutable `HashMap`** with a fixed set of key-value pairs

```java
Map<K, V> map = Map.of(key1, value1, key2, value2, ...);
Map<String, Integer> map = Map.of("Alice", 25, "Bob", 30);
```

6. Using `Map.ofEntries()` (Java 9+)

Creates an **immutable `HashMap`** with a fixed set of key-value pairs using `Map.entry()`.

Syntax

```java
Map<K, V> map = Map.ofEntries(
   Map.entry(key1, value1),
   Map.entry(key2, value2),
   ...
);

Map<String, Integer> map = Map.ofEntries(
   Map.entry("Alice", 25),
   Map.entry("Bob", 30)
);
```

7. Using Double Brace Initialization (Not Recommended)

Creates and initializes a `HashMap` in a single statement using an **anonymous inner class**.

Syntax

```
HashMap<K, V> map = new HashMap<>() {{
    put(key1, value1);
    put(key2, value2);
    ...
}};

HashMap<String, Integer> map = new HashMap<>() {{
    put("Alice", 25);
    put("Bob", 30);
}};
```

8. Using Streams (Java 8+)

Creates a `HashMap` from a stream of key-value pairs.

Syntax

```
HashMap<K, V> map = stream.collect(Collectors.toMap(...));

import java.util.stream.*;
import java.util.*;

public class HashMapExample {
    public static void main(String[] args) {
        HashMap<String, Integer> map = Stream.of(
            new AbstractMap.SimpleEntry<>("Alice", 25),
            new AbstractMap.SimpleEntry<>("Bob", 30)
        ).collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

        System.out.println(map); // Output: {Alice=25, Bob=30}
    }
}
```

Summary of Methods

Method	Description
Default Constructor	Creates an empty `HashMap` with default capacity (16) and load factor (0.75).
Constructor with Initial Capacity	Creates an empty `HashMap` with a specified initial capacity.
Constructor with Capacity and Load Factor	Creates an empty `HashMap` with custom capacity and load factor.
Constructor with Existing Map	Creates a `HashMap` and initializes it with an existing map.
`Map.of()` (Java 9+)	Creates an immutable `HashMap` with a fixed set of key-value pairs.
`Map.ofEntries()` (Java 9+)	Creates an immutable `HashMap` using `Map.entry()`.
Double Brace Initialization	Creates and initializes a `HashMap` using an anonymous inner class.
Using Streams (Java 8+)	Creates a `HashMap` from a stream of key-value pairs.

Key Points

1. **Default Constructor**: Use for most cases.
2. **Custom Capacity/Load Factor**: Use when you know the expected size of the map.
3. **Immutable Maps**: Use `Map.of()` or `Map.ofEntries()` for fixed, immutable maps.
4. **Streams**: Use for dynamic creation of maps from streams.

Let me know if you need further clarification or examples! 😊

A **HashMap** is a part of the **Java Collections Framework** and implements the **Map interface**. It stores data in **key-value pairs** and uses a **hashing technique** to store and retrieve values efficiently. Here's a detailed explanation, including **functions**, **example code**, and **important information**.

Key Features of HashMap

1. **Key-Value Pairs**: Stores data as key-value pairs.
2. **No Duplicate Keys**: Each key is unique. If a duplicate key is inserted, the old value is replaced.
3. **Allows Null Keys and Values**: A `HashMap` can have one `null` key and multiple `null` values.
4. **Unordered**: Does not maintain the order of insertion.
5. **Not Thread-Safe**: Not synchronized. For thread-safe operations, use `ConcurrentHashMap` or `Collections.synchronizedMap()`.

HashMap Functions

Here's a list of **important methods** in the `HashMap` class:

Method	Description

`put(K key, V value)`	Inserts a key-value pair into the map. If the key already exists, the value is replaced.
`get(Object key)`	Returns the value associated with the specified key. Returns `null` if the key is not found.
`remove(Object key)`	Removes the key-value pair for the specified key.
`containsKey(Object key)`	Returns `true` if the map contains the specified key.
`containsValue(Object value)`	Returns `true` if the map contains the specified value.
`size()`	Returns the number of key-value pairs in the map.
`isEmpty()`	Returns `true` if the map is empty.
`clear()`	Removes all key-value pairs from the map.
`keySet()`	Returns a `Set` of all keys in the map.
`values()`	Returns a `Collection` of all values in the map.
`entrySet()`	Returns a `Set` of all key-value pairs (entries) in the map.
`putAll(Map<? extends K, ? extends V> m)`	Copies all key-value pairs from the specified map to this map.
`getOrDefault(Object key, V defaultValue)`	Returns the value for the specified key, or a default value if the key is not found.
`replace(K key, V value)`	Replaces the value for the specified key only if it is currently mapped to some value.
`forEach(BiConsumer<? super K, ? super V> action)`	Performs the given action for each key-value pair in the map.

```
import java.util.HashMap;
import java.util.Map;
```

```
import java.util.function.BiConsumer;

public class MapMethodsDemo {
    public static void main(String[] args) {
        // Creating the main map
        Map<String, Integer> mainMap = new HashMap<>();
        mainMap.put("Apple", 50);
        mainMap.put("Banana", 20);
        mainMap.put("Orange", 30);

        // Creating another map to merge into mainMap
        Map<String, Integer> additionalMap = new HashMap<>();
        additionalMap.put("Grapes", 40);
        additionalMap.put("Mango", 60);

        // Using putAll() to copy all key-value pairs from additionalMap
        mainMap.putAll(additionalMap);

        // Using getOrDefault()
        int applePrice = mainMap.getOrDefault("Apple", 0); // Returns 50
        int watermelonPrice = mainMap.getOrDefault("Watermelon", 100); // Returns
default 100

        // Using replace()
        mainMap.replace("Banana", 25);       // Updates value to 25 if Banana exists
        mainMap.replace("Pineapple", 45);    // Won't do anything as Pineapple doesn't
exist

        // Using putIfAbsent()
        mainMap.putIfAbsent("Strawberry", 70);   // Adds Strawberry only if not present
        mainMap.putIfAbsent("Apple", 55);      // Won't change Apple's value since it
alreadyexists

        // Using forEach() to iterate and print each key-value pair
        mainMap.forEach(new BiConsumer<String, Integer>() {
            @Override
            public void accept(String key, Integer value) {
                System.out.println(key + " -> " + value);
            }
        });
```

```java
        // Using lambda expression for forEach()
        mainMap.forEach((key, value) -> System.out.println(key + " : " + value));
    }
}
```

```java
import java.util.HashMap;

public class HashMapExample {
    public static void main(String[] args) {
        // Create a HashMap
        HashMap<String, Integer> map = new HashMap<>();

        // Add key-value pairs
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Access a value using a key
        System.out.println("Age of Alice: " + map.get("Alice")); // Output: 25

        // Check if a key exists
        System.out.println("Contains key 'Bob': " + map.containsKey("Bob")); // Output:
true

        // Check if a value exists
        System.out.println("Contains value 35: " + map.containsValue(35)); // Output: true

        // Remove a key-value pair
        map.remove("Charlie");
        System.out.println("After removing Charlie: " + map); // Output: {Alice=25, Bob=30}

        // Iterate over the map
        System.out.println("Iterating over the map:");
        map.forEach((key, value) -> System.out.println(key + " -> " + value));

        // Get the size of the map
```

```
        System.out.println("Size of the map: " + map.size()); // Output: 2

        // Clear the map
        map.clear();
        System.out.println("Map after clearing: " + map); // Output: {}
    }
}
```

```
Age of Alice: 25
Contains key 'Bob': true
Contains value 35: true
After removing Charlie: {Alice=25, Bob=30}
Iterating over the map:
Alice -> 25
Bob -> 30
Size of the map: 2
Map after clearing: {}
```

Important Information

1. **Internal Working**:
   - `HashMap` uses a **hash table** to store key-value pairs.
   - It uses the `hashCode()` method of the key to determine the bucket (index) where the key-value pair should be stored.
   - In case of **hash collisions** (two keys having the same hash code), it uses a **linked list** or **balanced tree** (in Java 8+) to store multiple entries in the same bucket.
2. **Performance**:
   - **Average Time Complexity**:
     - `put()`, `get()`, `remove()`: O(1)
   - **Worst Case Time Complexity** (due to collisions):
     - `put()`, `get()`, `remove()`: O(n)
3. **Thread Safety**:
   - `HashMap` is **not thread-safe**. For thread-safe operations, use:

        - `ConcurrentHashMap`
        - `Collections.synchronizedMap(new HashMap<>())`
4. **Initial Capacity and Load Factor**:
    - **Initial Capacity**: The initial number of buckets in the hash table (default: 16).
    - **Load Factor**: The threshold at which the hash table is resized (default: 0.75).
    - Example: `HashMap<String, Integer> map = new HashMap<>(32, 0.8f);`
5. **Use Cases**:
    - Storing and retrieving data using unique keys.
    - Implementing caches or lookup tables.
    - Counting occurrences of elements (e.g., word frequency).

Key Points to Remember

1. **No Order**: `HashMap` does not maintain the order of insertion.
2. **Null Keys and Values**: Allows one `null` key and multiple `null` values.
3. **Efficiency**: Provides constant-time performance for basic operations (`put`, `get`, `remove`).
4. **Thread Safety**: Use `ConcurrentHashMap` or `Collections.synchronizedMap()` for thread-safe operations.

This explanation, example code, and important information are **easy to copy** into Word or any other document. Let me know if you need further clarification! 😊

keySet() Returns a Set of all keys in the map.
values() Returns a Collection of all values in the map.
entrySet() Returns a Set of all key-value pairs (entries) in the map. give me code for this

Here's the **example code** demonstrating the use of `keySet()`, `values()`, and `entrySet()` methods in a `HashMap`. These methods are used to retrieve **keys**, **values**, and **key-value pairs** (entries) from the map.

Example Code

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
import java.util.Collection;

public class HashMapMethodsExample {
    public static void main(String[] args) {
        // Create a HashMap
        HashMap<String, Integer> map = new HashMap<>();

        // Add key-value pairs
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // 1. keySet() - Get all keys in the map
        Set<String> keys = map.keySet();
        System.out.println("Keys in the map: " + keys); // Output: [Alice, Bob, Charlie]

        // 2. values() - Get all values in the map
        Collection<Integer> values = map.values();
        System.out.println("Values in the map: " + values); // Output: [25, 30, 35]

        // 3. entrySet() - Get all key-value pairs (entries) in the map
        Set<Map.Entry<String, Integer>> entries = map.entrySet();
        System.out.println("Entries in the map:");
        for (Map.Entry<String, Integer> entry : entries) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
        // Output:
        // Alice -> 25
        // Bob -> 30
        // Charlie -> 35
    }
}
```

Keys in the map: [Alice, Bob, Charlie]
Values in the map: [25, 30, 35]
Entries in the map:
Alice -> 25
Bob -> 30
Charlie -> 35

Explanation of the Code

1. `keySet()`

- Returns a **`Set`** of all keys in the map.
- **Example**:

```
Set<String> keys = map.keySet();
System.out.println("Keys in the map: " + keys);
```

- Output: `[Alice, Bob, Charlie]`

2. `values()`

- Returns a **`Collection`** of all values in the map.
- **Example**:

```
Collection<Integer> values = map.values();
System.out.println("Values in the map: " + values);
```

Output: `[25, 30, 35]`

3. `entrySet()`

- Returns a **`Set`** of all key-value pairs (entries) in the map.
- Each entry is represented as a **`Map.Entry<K, V>`** object.
- **Example**:

```
Set<Map.Entry<String, Integer>> entries = map.entrySet();
for (Map.Entry<String, Integer> entry : entries) {
```

```
    System.out.println(entry.getKey() + " -> " +
entry.getValue());
}
```

```
Alice -> 25
Bob -> 30
Charlie -> 35
```

Key Points

1. **`keySet()`**: Use this to retrieve all keys in the map.
2. **`values()`**: Use this to retrieve all values in the map.
3. **`entrySet()`**: Use this to retrieve all key-value pairs (entries) in the map.
4. **Iteration**: You can use `entrySet()` to iterate over the map and access both keys and values.

This code is **ready to copy and paste** into your IDE or notebook. Let me know if you need further clarification! 😊

Great question! Let's break it down and address both parts of your query:

---

1. Can we use `List` instead of `Collection` for `map.values()`?

The `values()` method of a `HashMap` returns a **`Collection`**, not a `List`. However, you can **convert the `Collection` to a `List`** if needed. Here's how:

Example: Convert `Collection` to `List`

```
import java.util.*;

public class HashMapValuesExample {
    public static void main(String[] args) {
        // Create a HashMap
```

```
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Get values as a Collection
        Collection<Integer> valuesCollection = map.values();

        // Convert Collection to List
        List<Integer> valuesList = new
ArrayList<>(valuesCollection);

        // Print the List
        System.out.println("Values in the map (as List): " +
valuesList);
    }
}
```

```
Values in the map (as List): [25, 30, 35]
```

2. Can we store the values in another `HashMap`?

Yes, you can store the values in another `HashMap`. However, since `HashMap` stores **key-value pairs**, you need to decide what keys to use for the new map. Here's an example:

Example: Store Values in Another HashMap

```
import java.util.*;

public class HashMapStoreValuesExample {
    public static void main(String[] args) {
        // Create the original HashMap
        HashMap<String, Integer> originalMap = new HashMap<>();
        originalMap.put("Alice", 25);
```

```
        originalMap.put("Bob", 30);
        originalMap.put("Charlie", 35);

        // Create a new HashMap to store values
        HashMap<String, Integer> newMap = new HashMap<>();

        // Iterate over the original map and store values in the
new map
        for (Map.Entry<String, Integer> entry :
originalMap.entrySet()) {
            // Use the same keys or generate new keys
            newMap.put(entry.getKey(), entry.getValue());
        }

        // Print the new map
        System.out.println("New HashMap: " + newMap);
    }
}
```

```
New HashMap: {Alice=25, Bob=30, Charlie=35}
```

Key Points

1. **`map.values()`** returns a `Collection`, not a `List`. You can convert it to a `List` using `new ArrayList<>(map.values())`.
2. To store values in another `HashMap`, you need to decide what keys to use. You can reuse the original keys or generate new ones.
3. If you want to store only the values (without keys), consider using a `List` or `Set` instead of a `HashMap`.

When to Use What?

- **Use `List`:**
    - When you only need the values and want to maintain their order or perform list-specific operations (e.g., sorting, indexing).
- **Use `HashMap`:**
    - When you need to store key-value pairs, and you have a meaningful way to assign keys to the values.

Let me know if you need further clarification or additional examples! 😊

ArrayList<Integer> valuesCollection = map.values(); ,HashMap<String, Integer> entries = map.entrySet(); i want this what is the problem

The issue with the code snippets you provided is that **`map.values()`** and **`map.entrySet()`** return specific types that cannot be directly assigned to an `ArrayList` or `HashMap`. Let's break it down and explain why these assignments are problematic, and how to fix them.

---

1. `map.values()` Returns a `Collection`, Not an `ArrayList`

The `values()` method of a `HashMap` returns a **`Collection<V>`**, not an `ArrayList<V>`. A `Collection` is a more general interface, and you cannot directly assign it to an `ArrayList`.

Problematic Code

```
ArrayList<Integer> valuesCollection = map.values(); // Error!
```

Why It Fails

- `map.values()` returns a `Collection<Integer>`, which is a supertype of `ArrayList<Integer>`.
- You cannot directly assign a supertype (`Collection`) to a subtype (`ArrayList`).

Solution

You need to **convert the `Collection` to an `ArrayList`** explicitly. Here's how:

```
// Correct way to convert Collection to ArrayList
ArrayList<Integer> valuesList = new ArrayList<>(map.values());
```

2. `map.entrySet()` Returns a `Set`, Not a `HashMap`

The `entrySet()` method of a `HashMap` returns a **`Set<Map.Entry<K, V>>`**, not a `HashMap<K, V>`. A `Set<Map.Entry<K, V>>` contains key-value pairs as `Map.Entry` objects, not as a `HashMap`.

Problematic Code

```
HashMap<String, Integer> entries = map.entrySet(); // Error!
```

Why It Fails

- `map.entrySet()` returns a `Set<Map.Entry<String, Integer>>`, which is a collection of key-value pairs.
- You cannot directly assign a `Set<Map.Entry<K, V>>` to a `HashMap<K, V>`.

Solution

If you want to create a new `HashMap` from the entries, you need to **iterate over the `entrySet()`** and populate the new `HashMap`. Here's how:

```
// Correct way to create a new HashMap from entries
HashMap<String, Integer> newMap = new HashMap<>();
for (Map.Entry<String, Integer> entry : map.entrySet()) {
    newMap.put(entry.getKey(), entry.getValue());
}
```

Full Example Code

Here's the corrected and complete code:

```
import java.util.*;
```

```java
public class HashMapExample {
    public static void main(String[] args) {
        // Create a HashMap
        HashMap<String, Integer> map = new HashMap<>();
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // 1. Convert map.values() to ArrayList
        ArrayList<Integer> valuesList = new
ArrayList<>(map.values());
        System.out.println("Values as ArrayList: " +
valuesList);

        // 2. Create a new HashMap from map.entrySet()
        HashMap<String, Integer> newMap = new HashMap<>();
        for (Map.Entry<String, Integer> entry : map.entrySet())
{
            newMap.put(entry.getKey(), entry.getValue());
        }
        System.out.println("New HashMap: " + newMap);
    }
}
Values as ArrayList: [25, 30, 35]
New HashMap: {Alice=25, Bob=30, Charlie=35}
```

Key Points

1. `map.values()` returns a `Collection<V>`. To convert it to an `ArrayList<V>`, use:

```java
ArrayList<V> valuesList = new ArrayList<>(map.values());
```

`map.entrySet()` returns a `Set<Map.Entry<K, V>>`. To create a new `HashMap<K, V>` from it, iterate over the entries and populate the new map:

```
HashMap<K, V> newMap = new HashMap<>();
for (Map.Entry<K, V> entry : map.entrySet()) {
   newMap.put(entry.getKey(), entry.getValue());
}
```

1. Why the Errors Occurred:
    - You cannot directly assign a `Collection` to an `ArrayList` because `Collection` is a supertype.
    - You cannot directly assign a `Set<Map.Entry<K, V>>` to a `HashMap<K, V>` because they are fundamentally different types.

Let me know if you need further clarification or additional examples! 😊

Let’s dive into the hash function, how elements are stored in a HashMap, what collisions are, how they are resolved, and the time complexity of HashMap operations.

---

1. Hash Function

A hash function is a function that takes an input (or "key") and returns a fixed-size string of bytes. The output is typically a hash code, which is an integer.

Purpose of a Hash Function

- Maps keys to indices in an array (called buckets or slots).
- Ensures that keys are distributed uniformly across the array to minimize collisions.

How Hash Functions Work in HashMap

- In Java, the `hashCode()` method of the key object is used to generate the hash code.
- The hash code is then processed further to ensure it fits within the bounds of the array:
-

```
int index = (hashCode & 0x7FFFFFFF) % arraySize;
```

- `0x7FFFFFFF` ensures the hash code is non-negative.
- `% arraySize` ensures the index is within the array bounds.

---

2. How Elements Are Stored in a HashMap

A HashMap internally uses an array of buckets to store key-value pairs. Here's how it works:

Internal Structure

- Array of Buckets: Each bucket can store one or more key-value pairs.
- Node: Each key-value pair is stored as a `Node` object (or `TreeNode` in Java 8+ for balanced trees).

Steps to Store an Element

1. Compute Hash Code: Use the `hashCode()` method of the key to compute the hash code.
2. Calculate Index: Use the hash code to calculate the index in the array:

```
int index = (hashCode & 0x7FFFFFFF) % arraySize;
```

1. Store in Bucket:
   - If the bucket is empty, store the key-value pair directly.
   - If the bucket is not empty, handle collisions (explained below).

---

3. Collisions in HashMap

A collision occurs when two or more keys produce the same hash code and are mapped to the same bucket.

How Collisions Are Resolved

1. Separate Chaining:
   - Each bucket is a linked list (or a balanced tree in Java 8+).
   - If a collision occurs, the new key-value pair is added to the linked list or tree in the same bucket.
   - Java 8+ Optimization: If a bucket has more than 8 elements, it converts the linked list into a balanced tree for faster lookups (O(log n) instead of O(n)).

2. **Open Addressing:**
   - **Not used in Java's `HashMap`, but some other hash table implementations use it.**
   - **If a collision occurs, the algorithm probes for the next available slot in the array.**

---

4. Time Complexity of HashMap Operations

Here's the time complexity of common HashMap operations:

Operation	Average Case	Worst Case
Insert (`put`)	O(1)	O(n) or O(log n)
Lookup (`get`)	O(1)	O(n) or O(log n)
Delete (`remove`)	O(1)	O(n) or O(log n)
Search (`containsKey`)	O(1)	O(n) or O(log n)

Explanation of Time Complexity

- Average Case (O(1)):
  - **Assumes a good hash function that distributes keys uniformly across buckets.**
  - **Collisions are rare, so most operations take constant time.**
- Worst Case (O(n) or O(log n)):
  - **Occurs when all keys collide and are stored in the same bucket.**
  - **In Java 8+, if a bucket has more than 8 elements, it uses a balanced tree, reducing the worst-case time complexity to O(log n).**

---

5. Example of HashMap Internal Working

Step-by-Step Example

1. **Insertion:**
   - Key: `"Alice"`, Value: `25`
   - Hash code of `"Alice"`: `12345`
   - Index: `12345 % 16 = 9` (assuming array size is 16)
   - Store `("Alice", 25)` at index 9.
2. **Collision Handling:**
   - Key: `"Bob"`, Value: `30`
   - Hash code of `"Bob"`: `12345` (same as `"Alice"`)
   - Index: `12345 % 16 = 9` (collision occurs)
   - Store `("Bob", 30)` in the same bucket as a linked list or tree.
3. **Lookup:**
   - Key: `"Alice"`
   - Hash code: `12345`
   - Index: `9`
   - Traverse the linked list or tree at index 9 to find `"Alice"`.

---

6. Key Points

1. **Hash Function:** Maps keys to indices in the array.
2. **Collisions:** Occur when two keys produce the same hash code.
3. **Collision Resolution:**
   - **Separate Chaining:** Uses linked lists or balanced trees.
   - **Java 8+ Optimization:** Converts linked lists to balanced trees for large buckets.
4. **Time Complexity:**
   - **Average Case:** O(1) for all operations.
   - **Worst Case:** O(n) or O(log n) due to collisions.

This explanation is clear and concise, and you can use it to understand how HashMap works internally. Let me know if you need further clarification! 😊

Hash function will give the output all ways same for the same key

A LinkedHashMap in Java is a subclass of `HashMap` that maintains a linked list of the entries in the map. This linked list defines the iteration order, which can be either:

1. Insertion Order: The order in which entries were inserted into the map (default behavior).
2. Access Order: The order in which entries were last accessed (from least-recently accessed to most-recently accessed).

In this explanation, we'll focus on access order and provide a practical example to demonstrate how it works.

What is Access Order in LinkedHashMap?

When a `LinkedHashMap` is configured to use access order, the order of the entries changes whenever an entry is accessed (via `get()` or `put()`). The most recently accessed entry is moved to the end of the linked list, making it the last entry in the iteration order.

Key Points:

- Access Order: Least-recently accessed entries are at the front, and most-recently accessed entries are at the end.
- Use Case: Useful for implementing LRU (Least Recently Used) caches.

How to Enable Access Order

To enable access order, you need to:

1. Use the `LinkedHashMap` constructor that accepts an initial capacity, load factor, and an access order flag.
2. Set the access order flag to `true`.

```
LinkedHashMap<K, V> map = new LinkedHashMap<>(initialCapacity, loadFactor, true);
```

Practical Example: Access Order in LinkedHashMap

Let's create a `LinkedHashMap` with access order enabled and demonstrate how the order changes when entries are accessed.

```
import java.util.LinkedHashMap;

public class LinkedHashMapAccessOrderExample {
    public static void main(String[] args) {
        // Create a LinkedHashMap with access order enabled
        LinkedHashMap<String, Integer> map = new LinkedHashMap<>(16, 0.75f, true);

        // Add entries to the map
        map.put("Alice", 25);
        map.put("Bob", 30);
        map.put("Charlie", 35);

        // Print the initial order of the map
        System.out.println("Initial Order: " + map); // {Alice=25, Bob=30, Charlie=35}

        // Access an entry (Bob)
        System.out.println("Value for Bob: " + map.get("Bob")); // 30

        // Print the order after accessing Bob
        System.out.println("Order after accessing Bob: " + map); // {Alice=25, Charlie=35,
Bob=30}

        // Access another entry (Alice)
        System.out.println("Value for Alice: " + map.get("Alice")); // 25

        // Print the order after accessing Alice
        System.out.println("Order after accessing Alice: " + map); // {Charlie=35, Bob=30,
Alice=25}

        // Add a new entry (David)
        map.put("David", 40);

        // Print the order after adding David
        System.out.println("Order after adding David: " + map); // {Charlie=35, Bob=30,
Alice=25, David=40}
    }
}
```

Explanation of the Example

1. **Initial Order:**
   - The entries are in the order they were inserted: `Alice`, `Bob`, `Charlie`.
2. **Accessing `Bob`:**
   - When `map.get("Bob")` is called, `Bob` is moved to the end of the linked list because it was accessed.
   - New order: `Alice`, `Charlie`, `Bob`.
3. **Accessing `Alice`:**
   - When `map.get("Alice")` is called, `Alice` is moved to the end of the linked list.
   - New order: `Charlie`, `Bob`, `Alice`.
4. **Adding `David`:**
   - When `map.put("David", 40)` is called, `David` is added to the end of the linked list.
   - New order: `Charlie`, `Bob`, `Alice`, `David`.

Key Points

1. **Access Order:**
   - The most recently accessed entry is moved to the end of the linked list.
   - This behavior is useful for implementing LRU caches.
2. **Insertion Order:**
   - If the access order flag is `false` (default), the iteration order is based on insertion order.
3. **Use Case:**
   - **LRU Cache**: Use `LinkedHashMap` with access order to automatically remove the least recently used entries when the cache reaches its capacity.

LRU Cache Example

Here's how you can use `LinkedHashMap` to implement a simple LRU Cache:

import java.util.LinkedHashMap;
import java.util.Map;

```
public class LRUCache<K, V> extends LinkedHashMap<K, V> {

 // here we are using capacity and giving value so that we can use in the remove eldest
//method

    private final int capacity;

    public LRUCache(int capacity) {
        super(capacity, 0.75f, true); // Enable access order
        this.capacity = capacity;
    }


//it is a overridden method and private capacity we get it from constructor if size >
//capacity remove least used element

    @Override
    protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {
        return size() > capacity; // Remove the eldest entry if the cache is full
    }

    public static void main(String[] args) {
        LRUCache<String, Integer> cache = new LRUCache<>(3);

        cache.put("Alice", 25);
        cache.put("Bob", 30);
        cache.put("Charlie", 35);

        System.out.println("Cache: " + cache); // {Alice=25, Bob=30, Charlie=35}

        cache.get("Alice"); // Access Alice
        System.out.println("Cache after accessing Alice: " + cache); // {Bob=30,
Charlie=35, Alice=25}

        cache.put("David", 40); // Add David (cache is full)
        System.out.println("Cache after adding David: " + cache); // {Charlie=35, Alice=25,
David=40}
    }
}
```

OUTPUT

Cache: {Alice=25, Bob=30, Charlie=35}
Cache after accessing Alice: {Bob=30, Charlie=35, Alice=25}
Cache after adding David: {Charlie=35, Alice=25, David=40}

Key Takeaways

1. **LinkedHashMap:**
   - **Maintains insertion or access order.**
   - **Slightly slower than `HashMap` due to linked list overhead.**
   - **Ideal for scenarios where order matters (e.g., LRU cache).**
2. **Interview Tips:**
   - **Understand the difference between `HashMap` and `LinkedHashMap`.**
   - **Be prepared to implement an LRU cache using `LinkedHashMap`.**

GARBAGE COLLECTOR

Object Graph in Java

The object graph is a conceptual representation of how objects in a Java program are interconnected through references. It is used by the Garbage Collector (GC) to determine which objects are reachable (still in use) and which are unreachable (eligible for garbage collection).

What is an Object Graph?

- **An object graph is a network of objects connected by references.**
- **Each object in the graph can reference other objects, forming a directed graph.**
- **The GC starts from GC roots and traverses the object graph to identify all reachable objects.**

Components of an Object Graph

1. **Objects:**
   - **Instances of classes or arrays.**
   - **Each object can hold references to other objects.**
2. **References:**
   - **Pointers from one object to another.**
   - **Examples: Fields, array elements, etc.**
3. **GC Roots:**
   - **Starting points for traversing the object graph.**
   - **Objects directly accessible by the JVM (e.g., local variables, static fields, active threads).**

How the Object Graph Works

1. GC Roots

The garbage collector starts traversing the object graph from the GC roots. Common GC roots include:

- **Local variables in the currently executing methods.**
- **Active threads.**
- **Static variables.**
- **JNI (Java Native Interface) references.**

2. Traversing the Graph

- **The GC traverses the object graph by following references from the GC roots to other objects.**
- **All objects reachable from the GC roots are marked as reachable.**
- **Objects not reachable from any GC root are marked as unreachable (eligible for garbage collection).**

3. Marking and Sweeping

- **Marking: The GC marks all reachable objects.**
- **Sweeping: The GC reclaims memory occupied by unreachable objects.**

Example of an Object Graph

Consider the following Java code:

```java
class Node {
    int value;
    Node next;

    public Node(int value) {
        this.value = value;
    }
}

public class ObjectGraphExample {
    public static void main(String[] args) {
        // Create objects
        Node node1 = new Node(10); // GC Root (local variable)
        Node node2 = new Node(20);
        Node node3 = new Node(30);

        // Create references
        node1.next = node2;
        node2.next = node3;

        // Remove reference to node2
        node2 = null;

        // Suggest JVM to perform garbage collection
        System.gc();
    }
}
```

Object Graph Representation

GC Root (node1) -> Node(10) -> Node(20) -> Node(30)

- **node1** is a GC root (local variable).
- **node1** references **node2**.
- **node2** references **node3**.

When `node2` is set to `null`, the reference chain is broken:

GC Root (node1) -> Node(10)

Node(20) -> Node(30) // Unreachable

- **node2** and **node3** are no longer reachable from any GC root.
- They are eligible for garbage collection.

How the Garbage Collector Uses the Object Graph

1. **Start from GC Roots**:
   - The GC starts traversing the object graph from the GC roots (e.g., `node1`).
2. **Mark Reachable Objects**:
   - The GC marks all objects reachable from the GC roots (e.g., `node1` and `node2` before `node2` is set to `null`).
3. **Identify Unreachable Objects**:
   - Objects not reachable from any GC root (e.g., `node2` and `node3` after `node2` is set to `null`) are marked as unreachable.
4. **Reclaim Memory**:
   - The GC reclaims memory occupied by unreachable objects.

Key Points

1. **Object Graph**:
   - A network of objects connected by references.
   - Used by the GC to determine reachability.
2. **GC Roots**:
   - Starting points for traversing the object graph.
   - Examples: Local variables, static fields, active threads.
3. **Reachability**:
   - Objects reachable from GC roots are considered in use.
   - Unreachable objects are eligible for garbage collection.
4. **Traversal**:
   - The GC traverses the object graph to mark reachable objects.

Interview Questions

1. What is an object graph, and how does it relate to garbage collection?
2. What are GC roots, and why are they important?
3. How does the garbage collector determine which objects are reachable?

4. Can you draw an object graph for a given piece of code?

Let me know if you need further clarification or additional examples! 😊

Garbage Collection in Java

Garbage Collection (GC) is a key feature of Java that automatically manages memory by reclaiming unused objects. It ensures that memory is efficiently utilized and prevents memory leaks. Here's a detailed explanation of how garbage collection works in Java.

What is Garbage Collection?

- **Definition**: Garbage Collection is the process of identifying and removing objects that are no longer in use by the application.
- **Purpose**: Free up memory occupied by unreachable objects to make it available for new objects.

Key Concepts

1. Reachability
An object is considered **reachable** if it can be accessed by any live thread through a chain of references. Objects that are not reachable are eligible for garbage collection.

2. Eligibility for Garbage Collection
An object becomes eligible for garbage collection when:

- It is no longer referenced by any live thread.
- All references to the object are explicitly set to `null`.
- It is part of an isolated island of objects (circular references with no external references).

3. Garbage Collection Roots
These are the starting points for determining reachability. Examples include:

- Local variables in the currently executing methods.
- Active threads.
- Static variables.
- JNI (Java Native Interface) references.

How Garbage Collection Works

1. Marking

- The garbage collector identifies all objects that are still in use (reachable) by traversing the object graph starting from the GC roots.
- All unreachable objects are marked as garbage.

2. Sweeping

- The garbage collector reclaims the memory occupied by unreachable objects.

3. Compacting (Optional)

- After sweeping, the garbage collector may move the remaining objects to consolidate free memory, reducing fragmentation.

Generational Garbage Collection

Java uses a **generational garbage collection** strategy, which divides the heap into different generations based on the age of objects:

1. Young Generation

- **Purpose**: Stores newly created objects.
- **Subdivisions**:
  - **Eden Space**: Where new objects are allocated.
  - **Survivor Spaces (S0 and S1)**: Objects that survive a minor GC are moved here.
- **Garbage Collection**: Minor GC (frequent and fast).

2. Old Generation

- **Purpose**: Stores long-lived objects that have survived multiple minor GCs.
- **Garbage Collection**: Major GC (less frequent but slower).

3. Permanent Generation (Java 7 and earlier) / Metaspace (Java 8+)

- **Purpose**: Stores metadata about classes and methods.
- **Garbage Collection**: Full GC (rare and slow).

Types of Garbage Collectors

Java provides several garbage collectors, each optimized for different use cases:

1. Serial GC

- **Use Case**: Single-threaded applications.
- **Algorithm**: Mark-Sweep-Compact.
- **Enabled By**: `-XX:+UseSerialGC`.

2. Parallel GC (Throughput Collector)

- **Use Case**: Multi-threaded applications with high throughput.
- **Algorithm**: Parallel version of Mark-Sweep-Compact.
- **Enabled By**: `-XX:+UseParallelGC`.

3. CMS (Concurrent Mark-Sweep) GC

- **Use Case**: Applications requiring low pause times.
- **Algorithm**: Concurrent Mark-Sweep.
- **Enabled By**: `-XX:+UseConcMarkSweepGC`.

4. G1 (Garbage-First) GC

- **Use Case**: Applications with large heaps and low pause time requirements.
- **Algorithm**: Divides the heap into regions and prioritizes garbage collection in regions with the most garbage.
- **Enabled By**: `-XX:+UseG1GC`.

5. ZGC (Z Garbage Collector)

- **Use Case**: Applications requiring very low pause times (sub-millisecond).
- **Algorithm**: Concurrent and scalable.
- **Enabled By**: `-XX:+UseZGC`.

6. Shenandoah GC

- **Use Case**: Applications requiring low pause times with large heaps.
- **Algorithm**: Concurrent and low-latency.
- **Enabled By**: `-XX:+UseShenandoahGC`.

Garbage Collection Process

1. **Object Creation**:
   - New objects are created in the **Eden Space** of the Young Generation.
2. **Minor GC**:
   - When the Eden Space is full, a **minor GC** is triggered.

    - Surviving objects are moved to the **Survivor Spaces** (S0 or S1).
3. **Promotion to Old Generation**:
    - Objects that survive multiple minor GCs are promoted to the **Old Generation**.
4. **Major GC**:
    - When the Old Generation is full, a **major GC** is triggered.
    - This is slower and involves reclaiming memory from the Old Generation.
5. **Full GC**:
    - Involves cleaning up both the Young and Old Generations.
    - Typically triggered when the heap is almost full.

Example of Garbage Collection in Action

```
public class GarbageCollectionExample {
    public static void main(String[] args) {
        // Create objects
        for (int i = 0; i < 100000; i++) {
            new GarbageCollectionExample();
        }

        // Suggest JVM to perform garbage collection
        System.gc();

        System.out.println("Garbage collection completed.");
    }

    @Override
    protected void finalize() throws Throwable {
        System.out.println("Object is being garbage collected.");
    }
}
```

```
Object is being garbage collected.
Object is being garbage collected.
...
Garbage collection completed.
```

Key Points

1. **Automatic Memory Management**: Java handles memory allocation and deallocation automatically.
2. **Generational GC**: Divides the heap into Young and Old Generations for efficient garbage collection.
3. **Garbage Collectors**: Choose the appropriate GC based on your application's requirements (e.g., low latency, high throughput).
4. **System.gc()**: Suggests JVM to perform garbage collection, but it's not guaranteed.

Interview Questions

1. What is garbage collection, and how does it work in Java?
2. What are the different generations in the Java heap?
3. Explain the difference between minor GC and major GC.
4. What are the different types of garbage collectors in Java?
5. How does the `finalize()` method work, and why is it deprecated?

Let me know if you need further clarification or additional examples! 😊

explain weak refernce to me

Weak References in Java

In Java, a **Weak Reference** is a type of reference that allows an object to be garbage collected even if it is referenced by a weak reference. Unlike strong references, weak references do not prevent the garbage collector from reclaiming the object. This makes them useful for scenarios where you want to maintain a reference to an object without preventing it from being garbage collected.

Key Concepts

1. Types of References in Java

Java provides four types of references, listed in order of strength:

1. **Strong Reference**: The default type of reference. Objects with strong references are not eligible for garbage collection.

Object obj = new Object(); // Strong reference

Soft Reference: Objects with soft references are only garbage collected when the JVM is running low on memory.

SoftReference<Object> softRef = new SoftReference<>(new Object());

Weak Reference: Objects with weak references are garbage collected as soon as no strong or soft references point to them.

WeakReference<Object> weakRef = new WeakReference<>(new Object());

Phantom Reference: Objects with phantom references are garbage collected, but the reference is enqueued before finalization.
PhantomReference<Object> phantomRef = new PhantomReference<>(new Object(), queue);

2. How Weak References Work

- A weak reference allows the garbage collector to reclaim the object it refers to.
- If an object is only referenced by weak references, it is eligible for garbage collection.
- Once the object is garbage collected, the weak reference is cleared (set to `null`).

Use Cases for Weak References

1. **Caching**:
    - Weak references are often used in caching mechanisms where you want to cache objects but allow them to be garbage collected when memory is low.
    - Example: `WeakHashMap` uses weak references for keys.
2. **Listeners and Callbacks**:

    - Weak references can be used to avoid memory leaks in listener or callback patterns by allowing unused listeners to be garbage collected.
3. **Metadata Storage**:
    - Weak references can be used to store metadata about objects without preventing those objects from being garbage collected.

Example: Using Weak References

Here's an example demonstrating how weak references work:

```
import java.lang.ref.WeakReference;

public class WeakReferenceExample {
    public static void main(String[] args) {
        // Create a strong reference to an object
        Object strongRef = new Object();

        // Create a weak reference to the same object
        WeakReference<Object> weakRef = new WeakReference<>(strongRef);

        // Check if the weak reference points to the object
        System.out.println("Weak reference before GC: " + weakRef.get()); // Output: java.lang.Object@<hashcode>

        // Suggest JVM to perform garbage collection
        System.gc();

        // Check if the weak reference still points to the object
        System.out.println("Weak reference after GC: " + weakRef.get()); // Output: null (if GC collected the object)
    }
}
```

```
Weak reference before GC: java.lang.Object@<hashcode>
Weak reference after GC: null
```

Explanation of the Example

1. **Strong Reference**:
   - `strongRef` is a strong reference to an object, so the object is not eligible for garbage collection.
2. **Weak Reference**:
   - `weakRef` is a weak reference to the same object.
   - If the object is only referenced by `weakRef`, it can be garbage collected.
3. **Garbage Collection**:
   - When `System.gc()` is called, the garbage collector reclaims the object because it is only referenced by a weak reference.
   - After garbage collection, `weakRef.get()` returns `null`.

WeakHashMap: A Practical Use Case

`WeakHashMap` is a special implementation of `Map` that uses weak references for keys. When a key is no longer strongly referenced, it is automatically removed from the map.

Example: Using WeakHashMap

```
import java.util.WeakHashMap;

public class WeakHashMapExample {
    public static void main(String[] args) {
        // Create a WeakHashMap
        WeakHashMap<Object, String> map = new WeakHashMap<>();

        // Create a key
        Object key = new Object();

        // Add a key-value pair to the map
        map.put(key, "Value");

        // Check if the key exists in the map
        System.out.println("Map before GC: " + map); // Output:
{java.lang.Object@<hashcode>=Value}

        // Remove the strong reference to the key
        key = null;
```

```
        // Suggest JVM to perform garbage collection
        System.gc();

        // Check if the key still exists in the map
        System.out.println("Map after GC: " + map); // Output: {}
    }
}
```

Map before GC: {java.lang.Object@<hashcode>=Value}
Map after GC: {}

Key Points

1. **Weak References**:
    - Allow objects to be garbage collected even if they are referenced.
    - Useful for caching, listeners, and metadata storage.
2. **WeakHashMap**:
    - A `Map` implementation that uses weak references for keys.
    - Automatically removes entries when keys are no longer strongly referenced.
3. **Garbage Collection**:
    - Weak references are cleared when the referenced object is garbage collected.

Interview Questions

1. What is a weak reference, and how does it differ from a strong reference?
2. What are the use cases for weak references?
3. How does `WeakHashMap` work internally?
4. Can weak references cause memory leaks? Why or why not?

Let me know if you need further clarification or additional examples! 😊

WEAKHASHMAP

WeakHashMap in Java

A **WeakHashMap** is a special implementation of the `Map` interface in Java that uses **weak references** for its keys. This means that the entries in a `WeakHashMap` can be automatically removed when the key is no longer strongly referenced elsewhere in the program. This behavior makes `WeakHashMap` particularly useful for **caching** and other scenarios where you want to associate metadata with objects without preventing those objects from being garbage collected.

Key Features of WeakHashMap

1. **Weak References for Keys**:
   - The keys in a `WeakHashMap` are stored as **weak references**.
   - If a key is no longer strongly referenced outside the `WeakHashMap`, it becomes eligible for garbage collection.
   - Once the key is garbage collected, the corresponding entry is automatically removed from the map.
2. **Automatic Cleanup**:
   - The `WeakHashMap` automatically removes entries when their keys are garbage collected.
   - This prevents memory leaks caused by holding onto unused keys.
3. **Not Thread-Safe**:
   - Like `HashMap`, `WeakHashMap` is not thread-safe. For concurrent access, use `Collections.synchronizedMap()` or `ConcurrentHashMap`.

How WeakHashMap Works

1. **Key Storage**:
   - Keys are stored as weak references (`WeakReference`).
   - Values are stored as strong references.
2. **Garbage Collection**:
   - When a key is no longer strongly referenced outside the `WeakHashMap`, it becomes eligible for garbage collection.
   - Once the key is garbage collected, the corresponding entry is removed from the map.
3. **Reference Queue**:

- The `WeakHashMap` uses a **reference queue** to track keys that have been garbage collected.
- When a key is garbage collected, it is added to the reference queue, and the `WeakHashMap` removes the corresponding entry.

Use Cases for WeakHashMap

1. **Caching**:
    - Use `WeakHashMap` to cache objects without preventing them from being garbage collected when they are no longer in use.
    - Example: Cache metadata about objects that may be short-lived.
2. **Listener or Callback Registries**:
    - Use `WeakHashMap` to store listeners or callbacks without causing memory leaks.
    - Example: Register listeners for events, but allow them to be garbage collected when no longer needed.
3. **Metadata Storage**:
    - Use `WeakHashMap` to store metadata about objects without preventing those objects from being garbage collected.
    - Example: Store additional information about objects in a large-scale application.

Example: Using WeakHashMap

Here's an example demonstrating how `WeakHashMap` works:

```
import java.util.WeakHashMap;

public class WeakHashMapExample {
    public static void main(String[] args) {
        // Create a WeakHashMap
        WeakHashMap<Object, String> map = new WeakHashMap<>();

        // Create a key
        Object key = new Object();

        // Add a key-value pair to the map
```

```
        map.put(key, "Value");

        // Check if the key exists in the map
        System.out.println("Map before GC: " + map); // Output:
{java.lang.Object@<hashcode>=Value}

        // Remove the strong reference to the key
        key = null;

        // Suggest JVM to perform garbage collection
        System.gc();

        // Check if the key still exists in the map
        System.out.println("Map after GC: " + map); // Output: {}
    }
}
```

```
Map before GC: {java.lang.Object@<hashcode>=Value}
Map after GC: {}
```

Explanation of the Example

1. **Key Creation**:
    - A strong reference (`key`) is created for an object.
2. **Adding to WeakHashMap**:
    - The key-value pair is added to the `WeakHashMap`.
3. **Removing Strong Reference**:
    - The strong reference to the key is set to `null`.
4. **Garbage Collection**:
    - When `System.gc()` is called, the key is garbage collected because it is no longer strongly referenced.
    - The `WeakHashMap` automatically removes the entry for the garbage-collected key.

Key Points

1. **Weak References**:
   - Keys in a `WeakHashMap` are stored as weak references.
   - Entries are automatically removed when keys are garbage collected.
2. **Automatic Cleanup**:
   - Prevents memory leaks by removing unused entries.
3. **Use Cases**:
   - Caching, listener registries, and metadata storage.
4. **Not Thread-Safe**:
   - Use `Collections.synchronizedMap()` or `ConcurrentHashMap` for thread-safe operations.

Interview Questions

1. What is a `WeakHashMap`, and how does it differ from a `HashMap`?
2. How does `WeakHashMap` handle garbage collection of keys?
3. What are the use cases for `WeakHashMap`?
4. Can `WeakHashMap` cause memory leaks? Why or why not?

Let me know if you need further clarification or additional examples! 😊

explain same example with strings as key with in method and out side method

In Java, **strings** behave differently when used as keys in a `WeakHashMap` compared to regular objects. This is because of **string interning** and the way the JVM handles string literals. Let's explore the same `WeakHashMap` example using strings as keys, both **inside a method** and **outside a method**, to understand how garbage collection works in these scenarios.

Key Points About Strings in Java

1. **String Interning**:
   - String literals (e.g., `"key"`) are **interned** by the JVM, meaning they are stored in a special memory area called the **string pool**.

    - Interned strings are strongly referenced by the JVM and are not eligible for garbage collection unless explicitly removed from the pool.
2. **String Objects**:
    - Strings created using the `new` keyword (e.g., `new String("key")`) are not interned and behave like regular objects.

Example 1: Using Strings Inside a Method

In this example, we use a **non-interned string** (created with `new String()`) as the key. Since the string is not interned, it behaves like a regular object and can be garbage collected.

Code

```
import java.util.WeakHashMap;

public class WeakHashMapStringExample {
    public static void main(String[] args) {
        // Create a WeakHashMap
        WeakHashMap<String, String> map = new WeakHashMap<>();

        // Create a non-interned string key inside the method
        String key = new String("key");

        // Add a key-value pair to the map
        map.put(key, "Value");

        // Check if the key exists in the map
        System.out.println("Map before GC: " + map); // Output: {key=Value}

        // Remove the strong reference to the key
        key = null;

        // Suggest JVM to perform garbage collection
        System.gc();
```

```
        // Check if the key still exists in the map
        System.out.println("Map after GC: " + map); // Output: {}
    }
}
```

Map before GC: {key=Value}
Map after GC: {}

Explanation

1. **Key Creation**:
    - The key is created using `new String("key")`, so it is **not interned**.
    - The key behaves like a regular object and is eligible for garbage collection.
2. **Adding to WeakHashMap**:
    - The key-value pair is added to the `WeakHashMap`.
3. **Removing Strong Reference**:
    - The strong reference to the key is set to `null`.
4. **Garbage Collection**:
    - When `System.gc()` is called, the key is garbage collected because it is no longer strongly referenced.
    - The `WeakHashMap` automatically removes the entry for the garbage-collected key.

Example 2: Using Strings Outside a Method

In this example, we use a **string literal** (e.g., `"key"`) as the key. Since string literals are **interned**, they are strongly referenced by the JVM and are not eligible for garbage collection.

Code

```
import java.util.WeakHashMap;

public class WeakHashMapStringExample {
    // String literal (interned)
    private static final String KEY = "key";

    public static void main(String[] args) {
        // Create a WeakHashMap
```

```
        WeakHashMap<String, String> map = new WeakHashMap<>();

        // Add a key-value pair to the map
        map.put(KEY, "Value");

        // Check if the key exists in the map
        System.out.println("Map before GC: " + map); // Output: {key=Value}

        // Suggest JVM to perform garbage collection
        System.gc();

        // Check if the key still exists in the map
        System.out.println("Map after GC: " + map); // Output: {key=Value}
    }
}
```

```
Map before GC: {key=Value}
Map after GC: {key=Value}
```

xplanation

1. **Key Creation**:
    - The key is a string literal (`"key"`), so it is **interned** by the JVM.
    - Interned strings are strongly referenced by the JVM and are not eligible for garbage collection.
2. **Adding to WeakHashMap**:
    - The key-value pair is added to the `WeakHashMap`.
3. **Garbage Collection**:
    - When `System.gc()` is called, the key is **not garbage collected** because it is interned and strongly referenced by the JVM.
    - The entry remains in the `WeakHashMap`.

Key Differences Between the Two Examples

Aspect	Inside Method (Non-Interned String)	Outside Method (Interned String)
Key Type	`new String("key")` (non-interned)	`"key"` (interned)
Eligible for GC	Yes	No
Behavior in WeakHashMap	Entry is removed after GC	Entry remains after GC

Key Takeaways

1. **Non-Interned Strings**:
   - Created using `new String()`.
   - Behave like regular objects and can be garbage collected.
   - Suitable for use as keys in `WeakHashMap`.
2. **Interned Strings**:
   - Created using string literals (e.g., `"key"`).
   - Strongly referenced by the JVM and not eligible for garbage collection.
   - Not suitable for use as keys in `WeakHashMap` if you want automatic cleanup.
3. **WeakHashMap Behavior**:
   - Automatically removes entries when keys are garbage collected.
   - Works as expected with non-interned strings but not with interned strings.

Interview Questions

1. What is the difference between interned and non-interned strings in Java?
2. How does `WeakHashMap` behave with interned strings as keys?
3. Why would you use `WeakHashMap` with non-interned strings?
4. Can `WeakHashMap` cause memory leaks with interned strings? Why or why not?

Let me know if you need further clarification or additional examples! 😊

Here’s a **smaller and simpler example** of using `WeakHashMap` with threads. This example demonstrates how entries are automatically removed from the `WeakHashMap` when keys are garbage collected, even in a multi-threaded environment.

Small Example: WeakHashMap with Threads

Code

```
import java.util.WeakHashMap;
import java.util.Map;

public class WeakHashMapSmallExample {
    public static void main(String[] args) throws InterruptedException {
        // Create a WeakHashMap
        Map<Object, String> map = new WeakHashMap<>();

        // Thread 1: Adds an entry to the map
        Thread writerThread = new Thread(() -> {
            Object key = new Object(); // Create a new key
            map.put(key, "Value");
            System.out.println("Added: " + key + " -> Value");
        });

        // Thread 2: Periodically checks the map and triggers GC
        Thread readerThread = new Thread(() -> {
            while (true) {
                System.out.println("Map contents: " + map);
                System.gc(); // Suggest garbage collection
                try {
                    Thread.sleep(1000); // Check the map every second
                } catch (InterruptedException e) {
                    break; // Exit the loop if interrupted
                }
            }
```

```
        });

        // Start both threads
        writerThread.start();
        readerThread.start();

        // Wait for the writer thread to finish
        writerThread.join();

        // Let the reader thread run for a while
        Thread.sleep(3000);

        // Stop the reader thread
        readerThread.interrupt();
    }
}
```

Added: java.lang.Object@<hashcode> -> Value
Map contents: {java.lang.Object@<hashcode>=Value}
Map contents: {}

Explanation

1. **Writer Thread**:
    - Creates a new key (an `Object`) and adds it to the `WeakHashMap` with a value.
    - The key is not strongly referenced outside the map.
2. **Reader Thread**:
    - Periodically prints the contents of the `WeakHashMap`.
    - Calls `System.gc()` to suggest garbage collection.
3. **Garbage Collection**:
    - Since the key is not strongly referenced outside the map, it becomes eligible for garbage collection.
    - When `System.gc()` is called, the key is garbage collected, and the corresponding entry is automatically removed from the `WeakHashMap`.
4. **Thread Safety**:
    - The example does not synchronize access to the `WeakHashMap`, so it is not thread-safe.

- In a real-world scenario, you should use `Collections.synchronizedMap()` or `ConcurrentHashMap` for thread-safe access.

Key Points

1. **Automatic Cleanup**:
   - Entries are automatically removed when keys are garbage collected.
2. **Thread Behavior**:
   - Threads can add or access entries in the `WeakHashMap`.
   - If a key is garbage collected while a thread is accessing the map, the entry will be removed.
3. **Thread Safety**:
   - Use `Collections.synchronizedMap()` or `ConcurrentHashMap` for concurrent access.

Thread-Safe Version

To make the `WeakHashMap` thread-safe, wrap it using `Collections.synchronizedMap()`:

```
Map<Object, String> map = Collections.synchronizedMap(new WeakHashMap<>());
```

This smaller example demonstrates the core behavior of `WeakHashMap` in a multi-threaded environment. Let me know if you need further clarification! 😊

IdentityHashMap in Java

An **IdentityHashMap** is a specialized implementation of the `Map` interface in Java that uses **reference equality** (`==`) instead of **object equality** (`equals()`) for comparing keys. This makes it different from `HashMap`, which uses `equals()` and `hashCode()` for key comparison.

Key Features of IdentityHashMap

1. **Reference Equality**:
    - Keys are compared using == (reference equality) instead of `equals()`.
    - Two keys are considered equal only if they refer to the **same object in memory**.
2. **No Hashing**:
    - Unlike `HashMap`, `IdentityHashMap` does not use the `hashCode()` method for key comparison.
    - It uses `System.identityHashCode()` to compute hash codes, which is based on the object's memory address.
3. **Use Cases**:
    - When you need to use objects as keys based on their **identity** (memory address) rather than their **state** (content).
    - Useful for scenarios like object serialization, caching, or maintaining metadata about objects.

Differences Between IdentityHashMap and HashMap

Feature	IdentityHashMap	HashMap
Key Comparison	Uses == (reference equality).	Uses `equals()` and `hashCode()` (object equality).
Hash Code Calculation	Uses `System.identityHashCode()`.	Uses `hashCode()` method of the key.
Performance	Slightly faster for certain use cases.	Slower for scenarios requiring reference equality.
Use Case	When keys must be compared by memory address.	When keys must be compared by content.

Example: IdentityHashMap vs HashMap

Let's compare the behavior of `IdentityHashMap` and `HashMap` using a simple example.

Code

```
import java.util.HashMap;
import java.util.IdentityHashMap;
import java.util.Map;

public class IdentityHashMapExample {
    public static void main(String[] args) {
        // Create two strings with the same content but different references
        String key1 = new String("key");
        String key2 = new String("key");

        // HashMap example
        Map<String, String> hashMap = new HashMap<>();
        hashMap.put(key1, "Value1");
        hashMap.put(key2, "Value2");

        System.out.println("HashMap:");
        System.out.println("key1 == key2: " + (key1 == key2)); // false
        System.out.println("key1.equals(key2): " + key1.equals(key2)); // true
        System.out.println("HashMap contents: " + hashMap); // {key=Value2}

        // IdentityHashMap example
        Map<String, String> identityHashMap = new IdentityHashMap<>();
        identityHashMap.put(key1, "Value1");
        identityHashMap.put(key2, "Value2");

        System.out.println("\nIdentityHashMap:");
        System.out.println("key1 == key2: " + (key1 == key2)); // false
        System.out.println("key1.equals(key2): " + key1.equals(key2)); // true
        System.out.println("IdentityHashMap contents: " + identityHashMap); //
{key=Value1, key=Value2}
    }
}
```

```
HashMap:
key1 == key2: false
key1.equals(key2): true
HashMap contents: {key=Value2}
```

```
IdentityHashMap:
key1 == key2: false
key1.equals(key2): true
IdentityHashMap contents: {key=Value1, key=Value2}
```

Explanation

1. **HashMap**:
   - Uses `equals()` and `hashCode()` for key comparison.
   - Since `key1` and `key2` have the same content (`"key"`), they are considered equal.
   - The second entry (`key2`) overwrites the first entry (`key1`).
2. **IdentityHashMap**:
   - Uses == for key comparison.
   - Since `key1` and `key2` are different objects in memory, they are considered different keys.
   - Both entries are stored in the map.

Practical Use Case: Object Metadata

`IdentityHashMap` is useful for storing metadata about objects without interfering with their equality or hash code.

Example: Storing Metadata

```
import java.util.IdentityHashMap;
import java.util.Map;

class Person {
    String name;

    public Person(String name) {
        this.name = name;
    }
}

public class IdentityHashMapMetadataExample {
    public static void main(String[] args) {
```

```
        // Create two Person objects with the same name
        Person person1 = new Person("Alice");
        Person person2 = new Person("Alice");

        // Create an IdentityHashMap to store metadata
        Map<Person, String> metadataMap = new IdentityHashMap<>();
        metadataMap.put(person1, "Metadata for Alice 1");
        metadataMap.put(person2, "Metadata for Alice 2");

        // Print the metadata map
        System.out.println("Metadata Map:");
        metadataMap.forEach((key, value) ->
            System.out.println(key.name + " -> " + value)
        );
    }
}
```

Explanation

1. **Person Objects**:
   - `person1` and `person2` have the same name (`"Alice"`) but are different objects in memory.
2. **IdentityHashMap**:
   - Treats `person1` and `person2` as different keys because they are different objects.
   - Stores metadata for both objects without overwriting.

Key Takeaways

1. **IdentityHashMap**:
   - Uses == for key comparison.
   - Suitable for scenarios where object identity (memory address) matters.
2. **HashMap**:
   - Uses `equals()` and `hashCode()` for key comparison.
   - Suitable for scenarios where object content matters.
3. **Use Cases**:
   - `IdentityHashMap`: Object metadata, caching, serialization.

- HashMap: General-purpose key-value storage.

Let me know if you need further clarification or additional examples! 😊

Got it! Let's modify the `Person` class to include **three fields**: `name`, `id`, and `password`. We'll then demonstrate how `HashMap` and `IdentityHashMap` behave with these objects. I'll also explain the process step by step.

Modified `Person` Class

The `Person` class now has three fields: `name`, `id`, and `password`. We'll override `equals()` and `hashCode()` to compare `Person` objects based on **all three fields**.

```
class Person {
    String name;
    int id;
    String password;

    public Person(String name, int id, String password) {
        this.name = name;
        this.id = id;
        this.password = password;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return id == person.id &&
               name.equals(person.name) &&
               password.equals(person.password);
    }

    @Override
```

```
    public int hashCode() {
        return Objects.hash(name, id, password);
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', id=" + id + ", password='" + password + "'}";
    }
}
```

Example with HashMap and IdentityHashMap

Now, let's create three `Person` objects with the same `name` but different `id` and `password`. We'll add them to both a `HashMap` and an `IdentityHashMap` and observe the behavior.

```
import java.util.HashMap;
import java.util.IdentityHashMap;
import java.util.Map;
import java.util.Objects;

public class HashMapIdentityHashMapExample {
    public static void main(String[] args) {
        // Create three Person objects with the same name but different id and password
        Person person1 = new Person("Alice", 1, "password1");
        Person person2 = new Person("Alice", 2, "password2");
        Person person3 = new Person("Alice", 3, "password3");

        // Create a HashMap to store metadata
        Map<Person, String> hashMap = new HashMap<>();
        hashMap.put(person1, "Metadata for Alice 1");
        hashMap.put(person2, "Metadata for Alice 2");
        hashMap.put(person3, "Metadata for Alice 3");

        // Print the HashMap
        System.out.println("HashMap:");
        hashMap.forEach((key, value) ->
            System.out.println(key + " -> " + value)
```

```
        );

        // Create an IdentityHashMap to store metadata
        Map<Person, String> identityHashMap = new IdentityHashMap<>();
        identityHashMap.put(person1, "Metadata for Alice 1");
        identityHashMap.put(person2, "Metadata for Alice 2");
        identityHashMap.put(person3, "Metadata for Alice 3");

        // Print the IdentityHashMap
        System.out.println("\nIdentityHashMap:");
        identityHashMap.forEach((key, value) ->
            System.out.println(key + " -> " + value)
        );
    }
}
```

```
HashMap:
Person{name='Alice', id=1, password='password1'} -> Metadata for Alice 1
Person{name='Alice', id=2, password='password2'} -> Metadata for Alice 2
Person{name='Alice', id=3, password='password3'} -> Metadata for Alice 3

IdentityHashMap:
Person{name='Alice', id=1, password='password1'} -> Metadata for Alice 1
Person{name='Alice', id=2, password='password2'} -> Metadata for Alice 2
Person{name='Alice', id=3, password='password3'} -> Metadata for Alice 3
```

Explanation

HashMap Behavior

1. **Key Comparison**:
    - `HashMap` uses the `equals()` and `hashCode()` methods to compare keys.
    - Since we overrode `equals()` and `hashCode()` in the `Person` class, `person1`, `person2`, and `person3` are considered **different keys** because their `id` and `password` fields are different.
2. **Result**:

- All three entries are added to the `HashMap` because the keys are unique based on their `id` and `password`.

IdentityHashMap Behavior

1. **Key Comparison**:
    - `IdentityHashMap` uses **reference equality** (==) to compare keys.
    - It does not use the `equals()` or `hashCode()` methods.
2. **Result**:
    - Since `person1`, `person2`, and `person3` are different objects in memory, they are treated as **different keys**.
    - All three entries are added to the `IdentityHashMap`.

Key Differences

Aspect	HashMap	IdentityHashMap
Key Comparison	Uses `equals()` and `hashCode()`.	Uses == (reference equality).
Behavior	Treats objects as equal if `equals()` is true.	Treats objects as equal only if they are the same object in memory.
Result with 3 Objects	All three entries are stored because `id` and `password` are different.	All three entries are stored because they are different objects in memory.

What Happens if `id` and `password` Are the Same?

If we create `Person` objects with the same `name`, `id`, and `password`, the behavior of `HashMap` and `IdentityHashMap` will differ.

Modified Example

```
Person person1 = new Person("Alice", 1, "password1");
Person person2 = new Person("Alice", 1, "password1"); // Same id and password
Person person3 = new Person("Alice", 1, "password1"); // Same id and password
```

HashMap:
Person{name='Alice', id=1, password='password1'} -> Metadata for Alice 3

IdentityHashMap:
Person{name='Alice', id=1, password='password1'} -> Metadata for Alice 1
Person{name='Alice', id=1, password='password1'} -> Metadata for Alice 2
Person{name='Alice', id=1, password='password1'} -> Metadata for Alice 3

Explanation

- **HashMap**:
  - Treats `person1`, `person2`, and `person3` as equal keys because their `name`, `id`, and `password` are the same.
  - Only the last entry (`person3`) remains in the `HashMap`.
- **IdentityHashMap**:
  - Treats `person1`, `person2`, and `person3` as different keys because they are different objects in memory.
  - All three entries are stored in the `IdentityHashMap`.

Summary

1. **HashMap**:
   - Compares keys using `equals()` and `hashCode()`.
   - Treats objects as equal if their `name`, `id`, and `password` are the same.
   - Overwrites entries if keys are equal.
2. **IdentityHashMap**:
   - Compares keys using == (reference equality).
   - Treats objects as equal only if they are the same object in memory.
   - Stores all entries even if the keys have the same content.

When to Use Which?

- Use **HashMap** when you want keys to be compared by their **content** (e.g., `name`, `id`, `password`).
- Use **IdentityHashMap** when you want keys to be compared by their **memory address**.

Let me know if you need further clarification! 😊

In hash map first check hash code if equal then check content

In identity hash map first hash code then location

TREE MAP

TreeMap in Java

`TreeMap` is a class in Java that implements the `NavigableMap` interface, which in turn extends the `SortedMap` interface. It stores key-value pairs in a sorted order based on the natural ordering of its keys or by a custom comparator provided at the time of creation.

Functions in TreeMap

Here are the key methods provided by `TreeMap`:

1. **Constructors:**
   - `TreeMap()`: Constructs a new, empty tree map, using the natural ordering of its keys.
   - `TreeMap(Comparator<? super K> comparator)`: Constructs a new, empty tree map, ordered according to the given comparator.
   - `TreeMap(Map<? extends K, ? extends V> m)`: Constructs a new tree map containing the same mappings as the given map, ordered according to the natural ordering of its keys.
   - `TreeMap(SortedMap<K, ? extends V> m)`: Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map.
2. **Basic Operations:**
   - `V put(K key, V value)`: Associates the specified value with the specified key in this map.
   - `V get(Object key)`: Returns the value to which the specified key is mapped, or `null` if this map contains no mapping for the key.
   - `V remove(Object key)`: Removes the mapping for this key from this TreeMap if present.

- `boolean containsKey(Object key)`: Returns `true` if this map contains a mapping for the specified key.
- `boolean containsValue(Object value)`: Returns `true` if this map maps one or more keys to the specified value.
- `int size()`: Returns the number of key-value mappings in this map.
- `boolean isEmpty()`: Returns `true` if this map contains no key-value mappings.
- `void clear()`: Removes all of the mappings from this map.

3. **Views:**
   - `Set<K> keySet()`: Returns a `Set` view of the keys contained in this map.
   - `Collection<V> values()`: Returns a `Collection` view of the values contained in this map.
   - `Set<Map.Entry<K, V>> entrySet()`: Returns a `Set` view of the mappings contained in this map.
4. **SortedMap Methods:**
   - `Comparator<? super K> comparator()`: Returns the comparator used to order the keys in this map, or `null` if this map uses the natural ordering of its keys.
   - `K firstKey()`: Returns the first (lowest) key currently in this map.
   - `K lastKey()`: Returns the last (highest) key currently in this map.
   - `SortedMap<K, V> headMap(K toKey)`: Returns a view of the portion of this map whose keys are strictly less than `toKey`.
   - `SortedMap<K, V> tailMap(K fromKey)`: Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
   - `SortedMap<K, V> subMap(K fromKey, K toKey)`: Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
5. **NavigableMap Methods:**
   - `Map.Entry<K, V> ceilingEntry(K key)`: Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
   - `K ceilingKey(K key)`: Returns the least key greater than or equal to the given key, or `null` if there is no such key.

  - `Map.Entry<K, V> floorEntry(K key)`: Returns a key-value mapping associated with the greatest key less than or equal to the given key, or `null` if there is no such key.
  - `K floorKey(K key)`: Returns the greatest key less than or equal to the given key, or `null` if there is no such key.
  - `Map.Entry<K, V> higherEntry(K key)`: Returns a key-value mapping associated with the least key strictly greater than the given key, or `null` if there is no such key.
  - `K higherKey(K key)`: Returns the least key strictly greater than the given key, or `null` if there is no such key.
  - `Map.Entry<K, V> lowerEntry(K key)`: Returns a key-value mapping associated with the greatest key strictly less than the given key, or `null` if there is no such key.
  - `K lowerKey(K key)`: Returns the greatest key strictly less than the given key, or `null` if there is no such key.
  - `Map.Entry<K, V> firstEntry()`: Returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
  - `Map.Entry<K, V> lastEntry()`: Returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
  - `Map.Entry<K, V> pollFirstEntry()`: Removes and returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
  - `Map.Entry<K, V> pollLastEntry()`: Removes and returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
  - `NavigableMap<K, V> descendingMap()`: Returns a reverse order view of the mappings contained in this map.
  - `NavigableSet<K> descendingKeySet()`: Returns a reverse order `NavigableSet` view of the keys contained in this map.
  - `NavigableMap<K, V> headMap(K toKey, boolean inclusive)`: Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
  - `NavigableMap<K, V> tailMap(K fromKey, boolean inclusive)`: Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`.

- `NavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)`: Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`

Explanation

- **SortedMap Methods:** These methods allow you to work with the map in a sorted manner, such as getting the first or last key, or getting a submap.
- **NavigableMap Methods:** These methods provide more advanced navigation capabilities, such as finding the closest match for a given key, or getting a reverse-order view of the map.

This example demonstrates the use of various methods provided by `TreeMap`, including those inherited from `SortedMap` and `NavigableMap`.

Explanation of `SortedMap` Methods with a Program

The `SortedMap` interface provides methods to work with maps in a sorted order. Below is a Java program that demonstrates the use of the `SortedMap` methods:

1. **`comparator()`**: Returns the comparator used to order the keys, or `null` if natural ordering is used.
2. **`firstKey()`**: Returns the first (lowest) key in the map.
3. **`lastKey()`**: Returns the last (highest) key in the map.
4. **`headMap(K toKey)`**: Returns a view of the portion of the map whose keys are strictly less than `toKey`.
5. **`tailMap(K fromKey)`**: Returns a view of the portion of the map whose keys are greater than or equal to `fromKey`.
6. **`subMap(K fromKey, K toKey)`**: Returns a view of the portion of the map whose keys range from `fromKey` (inclusive) to `toKey` (exclusive).

Program Example

import java.util.*;

```
public class SortedMapExample {
    public static void main(String[] args) {
        // Create a TreeMap (implements SortedMap)
        SortedMap<Integer, String> sortedMap = new TreeMap<>();

        // Add elements to the map
        sortedMap.put(10, "Apple");
        sortedMap.put(20, "Banana");
        sortedMap.put(30, "Cherry");
        sortedMap.put(40, "Date");
        sortedMap.put(50, "Elderberry");

        // Display the sorted map
        System.out.println("SortedMap: " + sortedMap);

        // 1. comparator()
        Comparator<? super Integer> comparator = sortedMap.comparator();
        System.out.println("Comparator used: " + (comparator == null ? "Natural Ordering" :
comparator));

        // 2. firstKey()
        System.out.println("First key: " + sortedMap.firstKey());

        // 3. lastKey()
        System.out.println("Last key: " + sortedMap.lastKey());

        // 4. headMap(K toKey)
        SortedMap<Integer, String> headMap = sortedMap.headMap(30);
        System.out.println("HeadMap (keys < 30): " + headMap);

        // 5. tailMap(K fromKey)
        SortedMap<Integer, String> tailMap = sortedMap.tailMap(30);
        System.out.println("TailMap (keys >= 30): " + tailMap);

        // 6. subMap(K fromKey, K toKey)
        SortedMap<Integer, String> subMap = sortedMap.subMap(20, 40);
        System.out.println("SubMap (keys >= 20 and < 40): " + subMap);
    }
}
```

SortedMap: {10=Apple, 20=Banana, 30=Cherry, 40=Date, 50=Elderberry}
Comparator used: Natural Ordering
First key: 10
Last key: 50
HeadMap (keys < 30): {10=Apple, 20=Banana}
TailMap (keys >= 30): {30=Cherry, 40=Date, 50=Elderberry}
SubMap (keys >= 20 and < 40): {20=Banana, 30=Cherry}

Explanation of the Program

1. **`comparator()`**:
   - Since we didn't provide a custom comparator, the `TreeMap` uses natural ordering (ascending order of keys).
   - The method returns `null`, indicating natural ordering is used.
2. **`firstKey()`**:
   - The smallest key in the map is `10`, so `firstKey()` returns `10`.
3. **`lastKey()`**:
   - The largest key in the map is `50`, so `lastKey()` returns `50`.
4. **`headMap(K toKey)`**:
   - The `headMap(30)` method returns a view of the map containing keys strictly less than `30`.
   - The result is `{10=Apple, 20=Banana}`.
5. **`tailMap(K fromKey)`**:
   - The `tailMap(30)` method returns a view of the map containing keys greater than or equal to `30`.
   - The result is `{30=Cherry, 40=Date, 50=Elderberry}`.
6. **`subMap(K fromKey, K toKey)`**:
   - The `subMap(20, 40)` method returns a view of the map containing keys from `20` (inclusive) to `40` (exclusive).
   - The result is `{20=Banana, 30=Cherry}`.

Key Points

- The `SortedMap` interface ensures that the keys are always sorted.
- The `headMap`, `tailMap`, and `subMap` methods return **views** of the original map. Changes to the original map are reflected in these views, and vice versa.
- If you modify the original map (e.g., add or remove keys), the views will automatically reflect those changes.

This program demonstrates how to use the `SortedMap` methods effectively to work with sorted data in a map.

```
import java.util.*;

public class NavigableMapExample {
    public static void main(String[] args) {
        // Create a TreeMap (implements NavigableMap)
        NavigableMap<Integer, String> navigableMap = new TreeMap<>();

        // Add elements to the map
        navigableMap.put(10, "Apple");
        navigableMap.put(20, "Banana");
        navigableMap.put(30, "Cherry");
        navigableMap.put(40, "Date");
        navigableMap.put(50, "Elderberry");

        // Display the NavigableMap
        System.out.println("NavigableMap: " + navigableMap);

        // 1. ceilingEntry(K key)
        Map.Entry<Integer, String> ceilingEntry = navigableMap.ceilingEntry(25);
        System.out.println("Ceiling Entry for key 25: " + ceilingEntry); // Correct: 30=Cherry

        // 2. ceilingKey(K key)
        Integer ceilingKey = navigableMap.ceilingKey(25);
        System.out.println("Ceiling Key for key 25: " + ceilingKey); // Correct: 30
```

```
// 3. floorEntry(K key)
Map.Entry<Integer, String> floorEntry = navigableMap.floorEntry(25);
System.out.println("Floor Entry for key 25: " + floorEntry); // Correct: 20=Banana

// 4. floorKey(K key)
Integer floorKey = navigableMap.floorKey(25);
System.out.println("Floor Key for key 25: " + floorKey); // Correct: 20

// 5. higherEntry(K key)
Map.Entry<Integer, String> higherEntry = navigableMap.higherEntry(30);
System.out.println("Higher Entry for key 30: " + higherEntry); // Correct: 40=Date

// 6. higherKey(K key)
Integer higherKey = navigableMap.higherKey(30);
System.out.println("Higher Key for key 30: " + higherKey); // Correct: 40

// 7. lowerEntry(K key)
Map.Entry<Integer, String> lowerEntry = navigableMap.lowerEntry(30);
System.out.println("Lower Entry for key 30: " + lowerEntry); // Correct: 20=Banana

// 8. lowerKey(K key)
Integer lowerKey = navigableMap.lowerKey(30);
System.out.println("Lower Key for key 30: " + lowerKey); // Correct: 20

// 9. firstEntry()
Map.Entry<Integer, String> firstEntry = navigableMap.firstEntry();
System.out.println("First Entry: " + firstEntry); // Correct: 10=Apple

// 10. lastEntry()
Map.Entry<Integer, String> lastEntry = navigableMap.lastEntry();
System.out.println("Last Entry: " + lastEntry); // Correct: 50=Elderberry

// 11. pollFirstEntry()
Map.Entry<Integer, String> polledFirstEntry = navigableMap.pollFirstEntry();
System.out.println("Polled First Entry: " + polledFirstEntry); // Correct: 10=Apple
System.out.println("NavigableMap after polling first entry: " + navigableMap); //
Correct: {20=Banana, 30=Cherry, 40=Date, 50=Elderberry}

// 12. pollLastEntry()
```

```
        Map.Entry<Integer, String> polledLastEntry = navigableMap.pollLastEntry();
        System.out.println("Polled Last Entry: " + polledLastEntry); // Correct: 50=Elderberry
        System.out.println("NavigableMap after polling last entry: " + navigableMap); //
Correct: {20=Banana, 30=Cherry, 40=Date}

        // 13. descendingMap()
        NavigableMap<Integer, String> descendingMap = navigableMap.descendingMap();
        System.out.println("Descending Map: " + descendingMap); // Correct: {40=Date,
30=Cherry, 20=Banana}

        // 14. descendingKeySet()
        NavigableSet<Integer> descendingKeySet = navigableMap.descendingKeySet();
        System.out.println("Descending KeySet: " + descendingKeySet); // Correct: [40, 30,
20]

        // 15. headMap(K toKey, boolean inclusive)
        NavigableMap<Integer, String> headMap = navigableMap.headMap(30, true);
        System.out.println("HeadMap (keys <= 30): " + headMap); // Correct: {20=Banana,
30=Cherry}

        // 16. tailMap(K fromKey, boolean inclusive)
        NavigableMap<Integer, String> tailMap = navigableMap.tailMap(30, true);
        System.out.println("TailMap (keys >= 30): " + tailMap); // Correct: {30=Cherry,
40=Date}

        // 17. subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)
        NavigableMap<Integer, String> subMap = navigableMap.subMap(20, true, 40, false);
        System.out.println("SubMap (keys >= 20 and < 40): " + subMap); // Correct:
{20=Banana, 30=Cherry}
    }
}
```

```
NavigableMap: {10=Apple, 20=Banana, 30=Cherry, 40=Date, 50=Elderberry}
Ceiling Entry for key 25: 30=Cherry
Ceiling Key for key 25: 30
Floor Entry for key 25: 20=Banana
Floor Key for key 25: 20
Higher Entry for key 30: 40=Date
```

Higher Key for key 30: 40
Lower Entry for key 30: 20=Banana
Lower Key for key 30: 20
First Entry: 10=Apple
Last Entry: 50=Elderberry
Polled First Entry: 10=Apple
NavigableMap after polling first entry: {20=Banana, 30=Cherry, 40=Date, 50=Elderberry}
Polled Last Entry: 50=Elderberry
NavigableMap after polling last entry: {20=Banana, 30=Cherry, 40=Date}
Descending Map: {40=Date, 30=Cherry, 20=Banana}
Descending KeySet: [40, 30, 20]
HeadMap (keys <= 30): {20=Banana, 30=Cherry}
TailMap (keys >= 30): {30=Cherry, 40=Date}
SubMap (keys >= 20 and < 40): {20=Banana, 30=Cherry}

Sure! Here's a clear explanation of each `NavigableMap` method used in your program, without explicitly pointing out corrections:

---

1. `ceilingEntry(K key)`

- **Description**: Returns the smallest key-value pair where the key is **greater than or equal to** the given key.
- **Example**: For `ceilingEntry(25)`, the result is `30=Cherry` because `30` is the smallest key >= `25`.

---

2. `ceilingKey(K key)`

- **Description**: Similar to `ceilingEntry`, but returns only the **key** (not the key-value pair).
- **Example**: For `ceilingKey(25)`, the result is `30` because `30` is the smallest key >= `25`.

---

3. `floorEntry(K key)`

- **Description**: Returns the largest key-value pair where the key is **less than or equal to** the given key.

- **Example**: For `floorEntry(25)`, the result is `20=Banana` because `20` is the largest key <= `25`.

---

4. `floorKey(K key)`

- **Description**: Similar to `floorEntry`, but returns only the **key** (not the key-value pair).
- **Example**: For `floorKey(25)`, the result is `20` because `20` is the largest key <= `25`.

---

5. `higherEntry(K key)`

- **Description**: Returns the smallest key-value pair where the key is **strictly greater than** the given key.
- **Example**: For `higherEntry(30)`, the result is `40=Date` because `40` is the smallest key > `30`.

---

6. `higherKey(K key)`

- **Description**: Similar to `higherEntry`, but returns only the **key** (not the key-value pair).
- **Example**: For `higherKey(30)`, the result is `40` because `40` is the smallest key > `30`.

---

7. `lowerEntry(K key)`

- **Description**: Returns the largest key-value pair where the key is **strictly less than** the given key.
- **Example**: For `lowerEntry(30)`, the result is `20=Banana` because `20` is the largest key < `30`.

---

8. `lowerKey(K key)`

- **Description**: Similar to `lowerEntry`, but returns only the **key** (not the key-value pair).

- **Example**: For `lowerKey(30)`, the result is `20` because `20` is the largest key < `30`.

---

9. `firstEntry()`

- **Description**: Returns the key-value pair with the **smallest key** in the map.
- **Example**: The result is `10=Apple` because `10` is the smallest key.

---

10. `lastEntry()`

- **Description**: Returns the key-value pair with the **largest key** in the map.
- **Example**: The result is `50=Elderberry` because `50` is the largest key.

---

11. `pollFirstEntry()`

- **Description**: Removes and returns the key-value pair with the **smallest key** in the map.
- **Example**: The result is `10=Apple`, and the map is updated to `{20=Banana, 30=Cherry, 40=Date, 50=Elderberry}`.

---

12. `pollLastEntry()`

- **Description**: Removes and returns the key-value pair with the **largest key** in the map.
- **Example**: The result is `50=Elderberry`, and the map is updated to `{20=Banana, 30=Cherry, 40=Date}`.

---

13. `descendingMap()`

- **Description**: Returns a **reverse-order view** of the map.
- **Example**: The result is `{40=Date, 30=Cherry, 20=Banana}`.

---

14. `descendingKeySet()`

- **Description**: Returns a **reverse-order set** of the keys in the map.
- **Example**: The result is `[40, 30, 20]`.

---

15. `headMap(K toKey, boolean inclusive)`

- **Description**: Returns a view of the portion of the map whose keys are **less than (or equal to, if `inclusive` is `true`)** the given key.
- **Example**: For `headMap(30, true)`, the result is `{20=Banana, 30=Cherry}`.

---

16. `tailMap(K fromKey, boolean inclusive)`

- **Description**: Returns a view of the portion of the map whose keys are **greater than (or equal to, if `inclusive` is `true`)** the given key.
- **Example**: For `tailMap(30, true)`, the result is `{30=Cherry, 40=Date}`.

---

17. `subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)`

- **Description**: Returns a view of the portion of the map whose keys range from `fromKey` to `toKey`. The `fromInclusive` and `toInclusive` parameters determine whether the range is inclusive or exclusive.
- **Example**: For `subMap(20, true, 40, false)`, the result is `{20=Banana, 30=Cherry}`.

Summary of Key Points:

- **`ceilingEntry` and `ceilingKey`**: Find the smallest key >= given key.
- **`floorEntry` and `floorKey`**: Find the largest key <= given key.
- **`higherEntry` and `higherKey`**: Find the smallest key > given key.
- **`lowerEntry` and `lowerKey`**: Find the largest key < given key.
- **`firstEntry` and `lastEntry`**: Retrieve the smallest and largest key-value pairs.
- **`pollFirstEntry` and `pollLastEntry`**: Remove and return the smallest and largest key-value pairs.
- **`descendingMap` and `descendingKeySet`**: Provide reverse-order views of the map and keys.
- **`headMap`, `tailMap`, and `subMap`**: Provide views of specific ranges of the map.

This explanation covers all the methods used in your program and their behavior. Let me know if you need further clarification!

HASHTABLE

HashTable in Java

A **HashTable** is a legacy class in Java that implements the `Map` interface. It stores key-value pairs and uses a hash table for storage. It is similar to `HashMap`, but with two key differences:

1. **Thread Safety**: HashTable is **synchronized**, meaning it is thread-safe. Multiple threads can safely access and modify a HashTable without causing data corruption.
2. **Null Keys and Values**: HashTable does **not allow null keys or values**, whereas HashMap allows one null key and multiple null values.

Why is HashTable Thread-Safe?

HashTable achieves thread safety by synchronizing all its methods. This means that only one thread can access or modify the HashTable at a time. While this ensures thread safety, it can lead to **performance overhead** in multi-threaded environments because threads must wait for their turn to access the HashTable.

Example: HashTable with Two Threads

Below is an example demonstrating how HashTable is thread-safe. Two threads are used to add and retrieve data from the HashTable concurrently.

Alternatives to HashTable

While HashTable is thread-safe, it is rarely used in modern Java applications due to its performance overhead. Here are some alternatives:

1. **`HashMap`**:
   - Not thread-safe by default.
   - Faster than HashTable because it is not synchronized.
   - Use `Collections.synchronizedMap(new HashMap<>())` to make it thread-safe.

2. **`ConcurrentHashMap`**:
    - Introduced in Java 5 as part of the `java.util.concurrent` package.
    - Provides better performance than HashTable in multi-threaded environments.
    - Uses fine-grained locking (lock striping) to allow concurrent read and write operations.
3. **`Collections.synchronizedMap()`**:
    - Wraps a `HashMap` or any other map to make it thread-safe.
    - Similar to HashTable in terms of thread safety but more flexible.
4. **`ConcurrentSkipListMap`**:
    - A thread-safe map that maintains keys in sorted order.
    - Suitable for scenarios where sorted keys are required.

When to Use HashTable?

HashTable is rarely used in modern Java applications. However, it can still be useful in scenarios where:

- You need a thread-safe map and are working with legacy code.
- You cannot use `ConcurrentHashMap` or other modern alternatives due to compatibility issues.

When Was HashTable Introduced?

HashTable was introduced in **Java 1.0** as part of the original Java Collections Framework. It is one of the oldest collection classes in Java.

Key Differences Between HashTable and HashMap

Feature	HashTable	HashMap
Thread Safety	Synchronized (thread-safe)	Not thread-safe
Null Keys/Values	Not allowed	Allowed (one null key, multiple null values)
Performance	Slower due to synchronization	Faster

Legacy	Yes (introduced in Java 1.0)	No (introduced in Java 1.2)

Conclusion

- **HashTable** is a thread-safe, legacy collection class in Java.
- It is rarely used today due to its performance overhead and the availability of better alternatives like `ConcurrentHashMap`.
- Use `ConcurrentHashMap` or `Collections.synchronizedMap()` for thread-safe maps in modern applications.
- HashTable is still relevant in legacy systems or when compatibility with older Java versions is required.

In this code, I'll add a `HashMap` and demonstrate how it behaves differently from `HashTable` in a multi-threaded environment. The key difference is that `HashMap` is **not thread-safe**, meaning it can lead to data corruption or inconsistencies when accessed by multiple threads simultaneously. On the other hand, `HashTable` is thread-safe due to its synchronized methods.

Updated Code with HashMap

```
import java.util.Hashtable;
import java.util.HashMap;

public class HashTableExample {
    public static void main(String[] args) {
        // Create a HashTable
        Hashtable<Integer, String> hashtable = new Hashtable<>();

        // Create a HashMap
        HashMap<Integer, String> hashMap = new HashMap<>();

        // Thread 1: Adds elements to the HashTable and HashMap
        Thread thread1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                hashtable.put(i, "Value" + i);
                hashMap.put(i, "Value" + i);
```

```
            System.out.println("Thread 1 added: " + i + " = Value" + i);
            try {
                Thread.sleep(100); // Simulate some delay
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

    // Thread 2: Reads elements from the HashTable and HashMap
    Thread thread2 = new Thread(() -> {
        for (int i = 0; i < 5; i++) {
            String hashtableValue = hashtable.get(i);
            String hashMapValue = hashMap.get(i);
            System.out.println("Thread 2 read from HashTable: " + i + " = " +
hashtableValue);
            System.out.println("Thread 2 read from HashMap: " + i + " = " + hashMapValue);
            try {
                Thread.sleep(100); // Simulate some delay
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });

    // Start both threads
    thread1.start();
    thread2.start();

    // Wait for threads to finish
    try {
        thread1.join();
        thread2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    // Final state of the HashTable and HashMap
    System.out.println("Final HashTable: " + hashtable);
```

```
        System.out.println("Final HashMap: " + hashMap);
    }
}
```

Explanation of Differences

1. Thread Safety

- **HashTable**: Thread-safe because all its methods are synchronized. Multiple threads can safely access and modify the HashTable without causing data corruption.
- **HashMap**: Not thread-safe. If multiple threads access or modify the HashMap concurrently, it can lead to data corruption or inconsistencies.

2. Null Keys and Values

- **HashTable**: Does not allow null keys or values. Attempting to insert a null key or value will throw a `NullPointerException`.
- **HashMap**: Allows one null key and multiple null values.

3. Performance

- **HashTable**: Slower due to synchronization overhead. Only one thread can access the HashTable at a time.
- **HashMap**: Faster because it is not synchronized. However, it is not safe for use in multi-threaded environments without external synchronization.

Expected Output

HashTable Output:
The output for the HashTable will be consistent and correct because it is thread-safe. For example:

```
Thread 1 added: 0 = Value0
Thread 2 read from HashTable: 0 = Value0
Thread 1 added: 1 = Value1
Thread 2 read from HashTable: 1 = Value1
Thread 1 added: 2 = Value2
Thread 2 read from HashTable: 2 = Value2
Thread 1 added: 3 = Value3
```

Thread 2 read from HashTable: 3 = Value3
Thread 1 added: 4 = Value4
Thread 2 read from HashTable: 4 = Value4
Final HashTable: {0=Value0, 1=Value1, 2=Value2, 3=Value3, 4=Value4}

HashMap Output:

The output for the HashMap may be inconsistent or incorrect because it is not thread-safe. For example:

Copy

Thread 1 added: 0 = Value0
Thread 2 read from HashMap: 0 = null
Thread 1 added: 1 = Value1
Thread 2 read from HashMap: 1 = null
Thread 1 added: 2 = Value2
Thread 2 read from HashMap: 2 = null
Thread 1 added: 3 = Value3
Thread 2 read from HashMap: 3 = null
Thread 1 added: 4 = Value4
Thread 2 read from HashMap: 4 = null
Final HashMap: {0=Value0, 1=Value1, 2=Value2, 3=Value3, 4=Value4}

In this case, Thread 2 may read `null` values from the HashMap because Thread 1 has not yet finished adding the values. This demonstrates the **lack of thread safety** in `HashMap`.

Making HashMap Thread-Safe

If you need a thread-safe alternative to `HashTable`, you can use:

1. **`Collections.synchronizedMap(new HashMap<>())`**:
    - Wraps the HashMap in a synchronized wrapper.
    - Example:

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

`ConcurrentHashMap`:

- Provides better performance than `HashTable` and `Collections.synchronizedMap`.
- Example:

Map<Integer, String> concurrentHashMap = new ConcurrentHashMap<>();

Key Takeaways

- **HashTable** is thread-safe but slower due to synchronization.
- **HashMap** is not thread-safe but faster.
- Use `ConcurrentHashMap` or `Collections.synchronizedMap` for thread-safe maps in modern applications.
- Always choose the appropriate map implementation based on your threading requirements.

CUNCURRENT HASHMAP

ConcurrentHashMap in Java

`ConcurrentHashMap` is a thread-safe implementation of the `Map` interface introduced in Java 5 as part of the `java.util.concurrent` package. It is designed to provide high concurrency and scalability in multi-threaded environments while maintaining thread safety. Unlike `HashTable`, which uses a single lock for the entire map, `ConcurrentHashMap` uses fine-grained locking (lock striping) to allow multiple threads to read and write concurrently without blocking each other unnecessarily.

Key Features of ConcurrentHashMap

1. **Thread Safety**:
    - `ConcurrentHashMap` is thread-safe and allows concurrent read and write operations.
    - It does not require external synchronization.
2. **High Concurrency**:
    - It uses fine-grained locking (lock striping) to allow multiple threads to operate on different parts of the map simultaneously.
3. **No Null Keys or Values**:

    - Like `HashTable`, `ConcurrentHashMap` does not allow null keys or values.
4. **Scalability**:
    - It is designed to scale well in multi-threaded environments, making it suitable for high-concurrency applications.
5. **Iterators**:
    - Iterators returned by `ConcurrentHashMap` are **weakly consistent**, meaning they reflect the state of the map at the time of creation and may not reflect subsequent modifications.

Internal Working of ConcurrentHashMap

1. Segmentation (Before Java 8):

- In Java 7 and earlier, `ConcurrentHashMap` used a **segmented locking mechanism**.
- The map was divided into multiple segments, and each segment had its own lock.
- This allowed multiple threads to operate on different segments concurrently.

2. Node-Based Locking (Java 8 and Later):

- In Java 8, the internal implementation was redesigned to use a **node-based locking mechanism**.
- Instead of segments, the map uses a table of `Node` objects, where each `Node` represents a key-value pair.
- Locks are applied at the individual node level, allowing even finer-grained concurrency.

3. CAS (Compare-And-Swap) Operations:

- `ConcurrentHashMap` uses **CAS (Compare-And-Swap)** operations for thread-safe updates without locking.
- CAS is an atomic operation that ensures that a value is updated only if it matches the expected value.

4. Concurrency Level:

- The `concurrencyLevel` parameter in the constructor determines the number of threads that can write to the map concurrently.
- A higher concurrency level allows more threads to operate concurrently but may increase memory usage.

5. Load Factor and Capacity:

- Like `HashMap`, `ConcurrentHashMap` uses a load factor to determine when to resize the map.
- The default load factor is `0.75`, and the default initial capacity is `16`.

Differences Between ConcurrentHashMap and HashTable

Feature	**ConcurrentHashMap**	**HashTable**
Thread Safety	Thread-safe with fine-grained locking	Thread-safe with a single lock
Concurrency	High concurrency (multiple threads can read/write concurrently)	Low concurrency (only one thread can access at a time)
Null Keys/Values	Does not allow null keys or values	Does not allow null keys or values
Performance	Faster in multi-threaded environments	Slower due to synchronization
Locking Mechanism	Fine-grained locking (node-based in Java 8)	Coarse-grained locking (single lock for the entire map)
Scalability	Highly scalable	Less scalable
Iterators	Weakly consistent	Fail-fast

Example: ConcurrentHashMap in Action

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        // Create a ConcurrentHashMap
        ConcurrentHashMap<Integer, String> concurrentHashMap = new
ConcurrentHashMap<>();
```

```
// Thread 1: Adds elements to the map
Thread thread1 = new Thread(() -> {
    for (int i = 0; i < 5; i++) {
        concurrentHashMap.put(i, "Value" + i);
        System.out.println("Thread 1 added: " + i + " = Value" + i);
        try {
            Thread.sleep(100); // Simulate some delay
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

// Thread 2: Reads elements from the map
Thread thread2 = new Thread(() -> {
    for (int i = 0; i < 5; i++) {
        String value = concurrentHashMap.get(i);
        System.out.println("Thread 2 read: " + i + " = " + value);
        try {
            Thread.sleep(100); // Simulate some delay
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
});

// Start both threads
thread1.start();
thread2.start();

// Wait for threads to finish
try {
    thread1.join();
    thread2.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}

// Final state of the ConcurrentHashMap
```

```
        System.out.println("Final ConcurrentHashMap: " + concurrentHashMap);
    }
}
```

```
Thread 1 added: 0 = Value0
Thread 2 read: 0 = Value0
Thread 1 added: 1 = Value1
Thread 2 read: 1 = Value1
Thread 1 added: 2 = Value2
Thread 2 read: 2 = Value2
Thread 1 added: 3 = Value3
Thread 2 read: 3 = Value3
Thread 1 added: 4 = Value4
Thread 2 read: 4 = Value4
Final ConcurrentHashMap: {0=Value0, 1=Value1, 2=Value2, 3=Value3, 4=Value4}
```

Key Takeaways

1. **Thread Safety**:
   - `ConcurrentHashMap` is thread-safe and allows concurrent read and write operations.
2. **High Concurrency**:
   - It uses fine-grained locking (node-based in Java 8) to allow multiple threads to operate concurrently.
3. **Performance**:
   - It is faster than `HashTable` in multi-threaded environments due to its efficient locking mechanism.
4. **Scalability**:
   - It is highly scalable and suitable for high-concurrency applications.
5. **Use Cases**:
   - Use `ConcurrentHashMap` when you need a thread-safe map with high concurrency and scalability.
   - Avoid `HashTable` in modern applications due to its performance overhead.

Alternatives to ConcurrentHashMap

1. **`Collections.synchronizedMap(new HashMap<>())`**:
   - Wraps a `HashMap` in a synchronized wrapper.
   - Less efficient than `ConcurrentHashMap` for high-concurrency scenarios.
2. **`ConcurrentSkipListMap`**:
   - A thread-safe map that maintains keys in sorted order.
   - Suitable for scenarios where sorted keys are required.

Conclusion

- `ConcurrentHashMap` is the preferred choice for thread-safe maps in modern Java applications.
- It provides high concurrency, scalability, and performance compared to `HashTable`.
- Use it when you need a thread-safe map for multi-threaded environments.

Node-Based Locking and CAS in `ConcurrentHashMap` (Java 8 and Later)

In Java 8, the internal implementation of `ConcurrentHashMap` was redesigned to use a **node-based locking mechanism** and **CAS (Compare-And-Swap)** operations for thread-safe updates. Let's break down these concepts and explain them with an example.

---

1. Node-Based Locking Mechanism

In Java 8, `ConcurrentHashMap` uses an array of `Node` objects to store key-value pairs. Each `Node` represents a key-value pair and is part of a linked list or a tree (in case of high collisions). Instead of locking the entire map or a segment, locks are applied at the **individual node level**. This allows multiple threads to operate on different nodes concurrently, improving performance and scalability.

Example of Node-Based Locking

Consider a `ConcurrentHashMap` with the following structure:

Index 0: Node1 -> Node2 -> Node3
Index 1: Node4 -> Node5
Index 2: Node6

- Each `Node` contains a key, value, and a reference to the next node.

- If two threads try to modify nodes at different indices (e.g., `Index 0` and `Index 1`), they can proceed concurrently because they are working on different nodes.
- If two threads try to modify the same node (e.g., both trying to update `Node1`), one thread will acquire the lock, and the other will wait.

---

2. CAS (Compare-And-Swap) Operations

CAS is an atomic operation used to update a value in a thread-safe manner without locking. It works as follows:

1. **Compare**: Check if the current value of a variable matches the expected value.
2. **Swap**: If the comparison is true, update the variable to the new value.
3. **Atomicity**: The entire operation is performed atomically, ensuring thread safety.

CAS is used in `ConcurrentHashMap` for operations like `put`, `replace`, and `remove` to ensure thread safety without locking.

Example: CAS in Action

Let's say we have a `ConcurrentHashMap` with the following key-value pair:

ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();
map.put(1, "Value1");

Now, suppose two threads (`Thread1` and `Thread2`) try to update the value for key 1 simultaneously.

Thread1:

- Wants to update the value for key 1 from `"Value1"` to `"NewValue1"`.

Thread2:

- Wants to update the value for key 1 from `"Value1"` to `"NewValue2"`.

CAS Steps:

1. **Thread1** reads the current value of key 1 (`"Value1"`).
2. **Thread1** uses CAS to check if the current value is still `"Value1"`. If true, it updates the value to `"NewValue1"`.
3. **Thread2** reads the current value of key 1 (`"Value1"`).

4. **Thread2** uses CAS to check if the current value is still `"Value1"`. If true, it updates the value to `"NewValue2"`.

Outcome:

- Only one thread will succeed in updating the value, depending on which thread performs the CAS operation first.
- The other thread will retry the operation until it succeeds.

Code Example: Node-Based Locking and CAS

Here's a simplified example to demonstrate how `ConcurrentHashMap` uses node-based locking and CAS:

```
import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapExample {
    public static void main(String[] args) {
        // Create a ConcurrentHashMap
        ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

        // Add initial key-value pairs
        map.put(1, "Value1");
        map.put(2, "Value2");

        // Thread 1: Updates value for key 1
        Thread thread1 = new Thread(() -> {
            boolean updated = map.replace(1, "Value1", "NewValue1");
            System.out.println("Thread 1 update successful: " + updated);
        });

        // Thread 2: Updates value for key 1
        Thread thread2 = new Thread(() -> {
            boolean updated = map.replace(1, "Value1", "NewValue2");
            System.out.println("Thread 2 update successful: " + updated);
        });

        // Start both threads
        thread1.start();
```

```java
        thread2.start();

        // Wait for threads to finish
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Final value for key 1
        System.out.println("Final value for key 1: " + map.get(1));
    }
}
```

```
Thread 1 update successful: true
Thread 2 update successful: false
Final value for key 1: NewValue1
```

Explanation of the Output

1. **Thread1** successfully updates the value for key 1 from `"Value1"` to `"NewValue1"` using CAS.
2. **Thread2** attempts to update the value for key 1 from `"Value1"` to `"NewValue2"`, but the CAS operation fails because the current value is no longer `"Value1"` (it was updated by Thread1).
3. The final value for key 1 is `"NewValue1"`.

Key Points

1. **Node-Based Locking**:
    - Locks are applied at the individual node level, allowing multiple threads to operate on different nodes concurrently.
    - This improves concurrency and scalability compared to `HashTable`.
2. **CAS (Compare-And-Swap)**:
    - CAS ensures thread-safe updates without locking.
    - It is used for operations like `put`, `replace`, and `remove`.
3. **Thread Safety**:

    - `ConcurrentHashMap` ensures thread safety while allowing high concurrency.
4. **Performance**:
    - The combination of node-based locking and CAS makes `ConcurrentHashMap` highly efficient in multi-threaded environments.

Conclusion

- `ConcurrentHashMap` uses **node-based locking** and **CAS** to achieve thread safety and high concurrency.
- Node-based locking allows multiple threads to operate on different parts of the map simultaneously.
- CAS ensures that updates are performed atomically without locking, improving performance.
- This makes `ConcurrentHashMap` a powerful and efficient choice for multi-threaded applications.

In the given scenario, when **Thread2** attempts to update the value for key 1 from `"Value1"` to `"NewValue2"`, the **CAS (Compare-And-Swap)** operation fails because the current value of key 1 is no longer `"Value1"` (it was updated to `"NewValue1"` by **Thread1**).

What Does Thread2 Do?

When the CAS operation fails, **Thread2** will **retry the operation** until it succeeds or until some other condition is met (e.g., a maximum number of retries). Here's what happens step-by-step:

1. **CAS Failure**:
    - Thread2 reads the current value of key 1 (which is now `"NewValue1"`).
    - Thread2 attempts to perform the CAS operation:
        - **Compare**: Check if the current value is `"Value1"`.
        - **Swap**: If the comparison is true, update the value to `"NewValue2"`.
    - Since the current value is `"NewValue1"`, the comparison fails, and the CAS operation does not update the value.
2. **Retry Logic**:
    - Thread2 will retry the operation by reading the current value again and performing the CAS operation with the updated expected value.

    - In this case, Thread2 will now use `"NewValue1"` as the expected value and attempt to update it to `"NewValue2"`.
3. **Successful Update (If No Other Thread Interferes)**:
    - If no other thread modifies the value of key 1 during the retry, Thread2 will successfully update the value to `"NewValue2"`.

Updated Code with Retry Logic

Here's how the retry logic might look in code:

ConcurrentSkipListMap in Java

`ConcurrentSkipListMap` is a thread-safe implementation of the `NavigableMap` interface introduced in Java 6 as part of the `java.util.concurrent` package. It is based on a **skip list** data structure, which provides logarithmic-time complexity for most operations (e.g., `put`, `get`, `remove`) while maintaining thread safety. Unlike `TreeMap`, which is based on a **red-black tree**, `ConcurrentSkipListMap` is designed for high concurrency and scalability.

Key Features of ConcurrentSkipListMap

1. **Thread Safety**:
    - `ConcurrentSkipListMap` is thread-safe and allows concurrent read and write operations.
    - It does not require external synchronization.
2. **Sorted Keys**:
    - It maintains keys in **sorted order** (natural order or by a custom comparator).
3. **High Concurrency**:
    - It uses a **skip list** data structure, which allows multiple threads to operate on the map concurrently.
4. **No Null Keys or Values**:
    - Like `ConcurrentHashMap`, `ConcurrentSkipListMap` does not allow null keys or values.
5. **Iterators**:

- Iterators returned by `ConcurrentSkipListMap` are **weakly consistent**, meaning they reflect the state of the map at the time of creation and may not reflect subsequent modifications.

Internal Working of ConcurrentSkipListMap

1. Skip List Data Structure:

- A skip list is a probabilistic data structure that allows fast search, insertion, and deletion of elements.
- It consists of multiple layers of linked lists, where each layer is a subset of the layer below it.
- The bottom layer is a standard linked list containing all the elements, while the upper layers act as "express lanes" to skip over large portions of the list.

2. Search Process:

- To search for an element, the algorithm starts at the top-left corner of the skip list (the highest level).
- It moves right at the current level until it finds a key greater than or equal to the target key.
- It then moves down to the next level and repeats the process until it reaches the bottom level.
- This process ensures **logarithmic-time complexity** for search operations.

3. Insertion Process:

- To insert a new key-value pair, the algorithm first searches for the appropriate position in the skip list.
- It then inserts the new node at the bottom level and probabilistically adds it to higher levels to maintain the skip list structure.

4. Deletion Process:

- To delete a key-value pair, the algorithm searches for the node and removes it from all levels of the skip list.

Conclusion

- `ConcurrentSkipListMap` is a thread-safe, sorted map based on a skip list data structure.

- It provides high concurrency and scalability, making it suitable for multi-threaded environments.
- Use it when you need a thread-safe, sorted map with efficient concurrent operations.
- It is different from `TreeMap` in terms of thread safety, concurrency, and internal implementa

Internal Working of `ConcurrentSkipListMap` with Examples

Let's break down the internal working of `ConcurrentSkipListMap` using examples for **search**, **insertion**, and **deletion** operations. We'll also explain how the **skip list data structure** works.

---

1. Skip List Data Structure

A skip list is a probabilistic data structure that consists of multiple layers of linked lists. Each layer is a subset of the layer below it. The bottom layer is a standard linked list containing all the elements, while the upper layers act as "express lanes" to skip over large portions of the list.

Example of a Skip List:
Consider a skip list with the following keys: `1, 3, 5, 7, 9`.

Level 3: 1 -----------------------> 9
Level 2: 1 --------> 5 --------> 9
Level 1: 1 -> 3 -> 5 -> 7 -> 9

- **Level 1** (bottom layer): Contains all the keys in sorted order.
- **Level 2**: Contains a subset of keys (`1, 5, 9`), acting as an express lane.
- **Level 3**: Contains an even smaller subset of keys (`1, 9`), acting as a faster express lane.

---

2. Search Process

The search process starts at the top-left corner of the skip list (the highest level) and moves right until it finds a key greater than or equal to the target key. It then moves down to the next level and repeats the process until it reaches the bottom level.

Example: Searching for Key 5

1. **Start at Level 3**:
    - Compare 1 (current key) with 5 (target key).
    - Since $1 < 5$, move right to 9.
    - Since $9 > 5$, move down to Level 2.
2. **Move to Level 2**:
    - Compare 1 (current key) with 5.
    - Since $1 < 5$, move right to 5.
    - Since $5 == 5$, the key is found.
3. **Return the value** associated with key 5.

---

3. Insertion Process

To insert a new key-value pair, the algorithm first searches for the appropriate position in the skip list. It then inserts the new node at the bottom level and probabilistically adds it to higher levels to maintain the skip list structure.

Example: Inserting Key 4

1. **Search for the position**:
    - Start at Level 3:
        - Compare 1 with 4. Since $1 < 4$, move right to 9.
        - Since $9 > 4$, move down to Level 2.
    - Move to Level 2:
        - Compare 1 with 4. Since $1 < 4$, move right to 5.
        - Since $5 > 4$, move down to Level 1.
    - Move to Level 1:
        - Compare 1 with 4. Since $1 < 4$, move right to 3.
        - Since $3 < 4$, move right to 5.
        - Since $5 > 4$, the insertion position is between 3 and 5.
2. **Insert at Level 1**:
    - Insert 4 between 3 and 5 at Level 1.
3. **Probabilistically add to higher levels**:
    - Flip a coin to decide whether to add 4 to Level 2.
        - If heads, add 4 to Level 2 between 1 and 5.
    - Flip a coin again to decide whether to add 4 to Level 3.
        - If tails, stop.

Updated skip list:

Level 3: 1 ----------------------> 9
Level 2: 1 --------> 4 --------> 5 --------> 9
Level 1: 1 -> 3 -> 4 -> 5 -> 7 -> 9

4. Deletion Process

To delete a key-value pair, the algorithm searches for the node and removes it from all levels of the skip list.

Example: Deleting Key 5

1. **Search for the key**:
   - Start at Level 3:
     - Compare 1 with 5. Since 1 < 5, move right to 9.
     - Since 9 > 5, move down to Level 2.
   - Move to Level 2:
     - Compare 1 with 5. Since 1 < 5, move right to 4.
     - Since 4 < 5, move right to 5.
     - Since 5 == 5, the key is found.
2. **Remove from all levels**:
   - Remove 5 from Level 1, Level 2, and Level 3.

Updated skip list:

Level 3: 1 ----------------------> 9
Level 2: 1 --------> 4 --------> 9
Level 1: 1 -> 3 -> 4 -> 7 -> 9

Key Takeaways

1. **Skip List**:
   - A probabilistic data structure that allows fast search, insertion, and deletion with logarithmic-time complexity.
2. **Search Process**:
   - Starts at the top level and moves right/down to find the target key.
3. **Insertion Process**:
   - Inserts at the bottom level and probabilistically adds to higher levels.

4. **Deletion Process**:
   - Removes the key from all levels of the skip list.
5. **ConcurrentSkipListMap**:
   - A thread-safe, sorted map based on a skip list.
   - Suitable for high-concurrency scenarios where sorted keys are required.

A `ConcurrentSkipListMap` in Java is a concurrent, scalable map implementation that is sorted according to the natural ordering of its keys or by a specified comparator. It is part of the `java.util.concurrent` package and is thread-safe. Below is a list of the key methods available in `ConcurrentSkipListMap`:

Constructors

1. **ConcurrentSkipListMap()**
   Constructs a new, empty map, sorted according to the natural ordering of the keys.
2. **ConcurrentSkipListMap(Comparator<? super K> comparator)**
   Constructs a new, empty map, sorted according to the specified comparator.
3. **ConcurrentSkipListMap(Map<? extends K, ? extends V> m)**
   Constructs a new map containing the same mappings as the given map, sorted according to the natural ordering of the keys.
4. **ConcurrentSkipListMap(SortedMap<K, ? extends V> m)**
   Constructs a new map containing the same mappings and using the same ordering as the specified sorted map.

Basic Operations

5. **V put(K key, V value)**
   Associates the specified value with the specified key in this map.
6. **V get(Object key)**
   Returns the value to which the specified key is mapped, or `null` if the map contains no mapping for the key.
7. **V remove(Object key)**
   Removes the mapping for the specified key from this map if present.
8. **boolean containsKey(Object key)**
   Returns `true` if this map contains a mapping for the specified key.
9. **boolean containsValue(Object value)**
   Returns `true` if this map maps one or more keys to the specified value.

10. **int size()**
    Returns the number of key-value mappings in this map.
11. **boolean isEmpty()**
    Returns `true` if this map contains no key-value mappings.

Views

12. **Set<K> keySet()**
    Returns a `Set` view of the keys contained in this map.
13. **Collection<V> values()**
    Returns a `Collection` view of the values contained in this map.
14. **Set<Map.Entry<K, V>> entrySet()**
    Returns a `Set` view of the mappings contained in this map.

Submaps

15. **ConcurrentNavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)**
    Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`.
16. **ConcurrentNavigableMap<K, V> headMap(K toKey, boolean inclusive)**
    Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
17. **ConcurrentNavigableMap<K, V> tailMap(K fromKey, boolean inclusive)**
    Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`.
18. **ConcurrentNavigableMap<K, V> subMap(K fromKey, K toKey)**
    Equivalent to `subMap(fromKey, true, toKey, false)`.
19. **ConcurrentNavigableMap<K, V> headMap(K toKey)**
    Equivalent to `headMap(toKey, false)`.
20. **ConcurrentNavigableMap<K, V> tailMap(K fromKey)**
    Equivalent to `tailMap(fromKey, true)`.

Other Operations

21. **void clear()**
    Removes all of the mappings from this map.

22. **V putIfAbsent(K key, V value)**
    If the specified key is not already associated with a value, associates it with the given value.
23. **boolean remove(Object key, Object value)**
    Removes the entry for a key only if currently mapped to a given value.
24. **boolean replace(K key, V oldValue, V newValue)**
    Replaces the entry for a key only if currently mapped to a given value.
25. **V replace(K key, V value)**
    Replaces the entry for a key only if it is currently mapped to some value.
26. **Comparator<? super K> comparator()**
    Returns the comparator used to order the keys in this map, or `null` if this map uses the natural ordering of its keys.
27. **K firstKey()**
    Returns the first (lowest) key currently in this map.
28. **K lastKey()**
    Returns the last (highest) key currently in this map.
29. **Map.Entry<K, V> firstEntry()**
    Returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
30. **Map.Entry<K, V> lastEntry()**
    Returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
31. **Map.Entry<K, V> pollFirstEntry()**
    Removes and returns a key-value mapping associated with the least key in this map, or `null` if the map is empty.
32. **Map.Entry<K, V> pollLastEntry()**
    Removes and returns a key-value mapping associated with the greatest key in this map, or `null` if the map is empty.
33. **ConcurrentNavigableMap<K, V> descendingMap()**
    Returns a reverse order view of the mappings contained in this map.
34. **NavigableSet<K> descendingKeySet()**
    Returns a reverse order `NavigableSet` view of the keys contained in this map.
35. **NavigableSet<K> navigableKeySet()**
    Returns a `NavigableSet` view of the keys contained in this map.

Concurrent Operations

36. **V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)**

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.

37. **V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**
    Attempts to compute a mapping for the specified key and its current mapped value (or `null` if there is no current mapping).
38. **V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)**
    If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function.
39. **V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**
    If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

Bulk Operations

40. **void putAll(Map<? extends K, ? extends V> m)**
    Copies all of the mappings from the specified map to this map.

Miscellaneous

41. **Object clone()**
    Returns a shallow copy of this `ConcurrentSkipListMap` instance.
42. **boolean equals(Object o)**
    Compares the specified object with this map for equality.
43. **int hashCode()**
    Returns the hash code value for this map.
44. **String toString()**
    Returns a string representation of this map.

ConcurrentMap Interface Methods

45. **V getOrDefault(Object key, V defaultValue)**
    Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.
46. **void forEach(BiConsumer<? super K, ? super V> action)**
    Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.

47. **void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)**
    Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

NavigableMap Interface Methods

48. **Map.Entry<K, V> lowerEntry(K key)**
    Returns a key-value mapping associated with the greatest key strictly less than the given key, or `null` if there is no such key.
49. **K lowerKey(K key)**
    Returns the greatest key strictly less than the given key, or `null` if there is no such key.
50. **Map.Entry<K, V> floorEntry(K key)**
    Returns a key-value mapping associated with the greatest key less than or equal to the given key, or `null` if there is no such key.
51. **K floorKey(K key)**
    Returns the greatest key less than or equal to the given key, or `null` if there is no such key.
52. **Map.Entry<K, V> ceilingEntry(K key)**
    Returns a key-value mapping associated with the least key greater than or equal to the given key, or `null` if there is no such key.
53. **K ceilingKey(K key)**
    Returns the least key greater than or equal to the given key, or `null` if there is no such key.
54. **Map.Entry<K, V> higherEntry(K key)**
    Returns a key-value mapping associated with the least key strictly greater than the given key, or `null` if there is no such key.
55. **K higherKey(K key)**
    Returns the least key strictly greater than the given key, or `null` if there is no such key.

SortedMap Interface Methods

56. **SortedMap<K, V> subMap(K fromKey, K toKey)**
    Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
57. **SortedMap<K, V> headMap(K toKey)**
    Returns a view of the portion of this map whose keys are strictly less than `toKey`.

58. **SortedMap<K, V> tailMap(K fromKey)**
Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.

ConcurrentNavigableMap Interface Methods

59. **ConcurrentNavigableMap<K, V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)**
Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`.
60. **ConcurrentNavigableMap<K, V> headMap(K toKey, boolean inclusive)**
Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
61. **ConcurrentNavigableMap<K, V> tailMap(K fromKey, boolean inclusive)**
Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`.

Additional Methods

62. **void forEach(BiConsumer<? super K, ? super V> action)**
Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
63. **void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)**
Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.
64. **V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)**
If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
65. **V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**
Attempts to compute a mapping for the specified key and its current mapped value (or `null` if there is no current mapping).
66. **V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)**
If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function.
67. **V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**

If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

Batch 1: Constructors

```
import java.util.*;
import java.util.concurrent.ConcurrentSkipListMap;

public class ConcurrentSkipListMapExample {
    public static void main(String[] args) {
        // 1. Default constructor
        ConcurrentSkipListMap<Integer, String> map1 = new ConcurrentSkipListMap<>();
        System.out.println("Map1: " + map1);

        // 2. Constructor with Comparator

        ConcurrentSkipListMap<Integer, String> map2 = new
<>(Comparator.reverseOrder());
        map2.put(3, "Three");
        map2.put(1, "One");
        System.out.println("Map2 (Reverse Order): " + map2);

        // 3. Constructor with Map
        Map<Integer, String> tempMap = new HashMap<>();
        tempMap.put(2, "Two");
        tempMap.put(4, "Four");
        ConcurrentSkipListMap<Integer, String> map3 = new
ConcurrentSkipListMap<>(tempMap);

        System.out.println("Map3: " + map3);

        // 4. Constructor with SortedMap

        SortedMap<Integer, String> sortedMap = new TreeMap<>();
        sortedMap.put(5, "Five");
        sortedMap.put(6, "Six");
```

```
        ConcurrentSkipListMap<Integer, String> map4 = new
ConcurrentSkipListMap<>(sortedMap);
        System.out.println("Map4: " + map4);
    }
}
```

Batch 2: Basic Operations

```
import java.util.concurrent.ConcurrentSkipListMap;

public class BasicOperations {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();

        // 5. put(K key, V value)
        map.put(1, "One");
        map.put(2, "Two");
        System.out.println("After put: " + map);

        // 6. get(Object key)
        String value = map.get(1);
        System.out.println("Value for key 1: " + value);

        // 7. remove(Object key)
        map.remove(2);
        System.out.println("After remove: " + map);

        // 8. containsKey(Object key)
        boolean containsKey = map.containsKey(1);
        System.out.println("Contains key 1: " + containsKey);

        // 9. containsValue(Object value)
        boolean containsValue = map.containsValue("One");
        System.out.println("Contains value 'One': " + containsValue);

        // 10. size()
        int size = map.size();
        System.out.println("Size: " + size);
```

```java
        // 11. isEmpty()
        boolean isEmpty = map.isEmpty();
        System.out.println("Is empty: " + isEmpty);
    }
}
```

Batch 3: Views

```java
import java.util.concurrent.ConcurrentSkipListMap;

public class ViewsExample {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
        map.put(1, "One");
        map.put(2, "Two");
        map.put(3, "Three");

        // 12. keySet()
        System.out.println("KeySet: " + map.keySet());

        // 13. values()
        System.out.println("Values: " + map.values());

        // 14. entrySet()
        System.out.println("EntrySet: " + map.entrySet());
    }
}
```

Batch 4: Submaps

```java
import java.util.concurrent.ConcurrentNavigableMap;
import java.util.concurrent.ConcurrentSkipListMap;

public class SubmapsExample {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
```

```
        map.put(1, "One");
        map.put(2, "Two");
        map.put(3, "Three");
        map.put(4, "Four");

        // 15. subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)
        ConcurrentNavigableMap<Integer, String> subMap = map.subMap(2, true, 4, false);
        System.out.println("SubMap (2 inclusive, 4 exclusive): " + subMap);

        // 16. headMap(K toKey, boolean inclusive)
        ConcurrentNavigableMap<Integer, String> headMap = map.headMap(3, true);
        System.out.println("HeadMap (up to 3 inclusive): " + headMap);

        // 17. tailMap(K fromKey, boolean inclusive)
        ConcurrentNavigableMap<Integer, String> tailMap = map.tailMap(2, false);
        System.out.println("TailMap (from 2 exclusive): " + tailMap);

        // 18. subMap(K fromKey, K toKey)
        ConcurrentNavigableMap<Integer, String> subMap2 = map.subMap(1, 3);
        System.out.println("SubMap (1 inclusive, 3 exclusive): " + subMap2);

        // 19. headMap(K toKey)
        ConcurrentNavigableMap<Integer, String> headMap2 = map.headMap(3);
        System.out.println("HeadMap (up to 3 exclusive): " + headMap2);

        // 20. tailMap(K fromKey)
        ConcurrentNavigableMap<Integer, String> tailMap2 = map.tailMap(2);
        System.out.println("TailMap (from 2 inclusive): " + tailMap2);
    }
}
```

Batch 5: Other Operations

```
import java.util.concurrent.ConcurrentSkipListMap;

public class OtherOperations {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
```

```
map.put(1, "One");
map.put(2, "Two");

// 21. clear()
map.clear();
System.out.println("After clear: " + map);

// 22. putIfAbsent(K key, V value)
map.putIfAbsent(1, "One");
map.putIfAbsent(1, "New One");
System.out.println("After putIfAbsent: " + map);

// 23. remove(Object key, Object value)
boolean removed = map.remove(1, "One");
System.out.println("Removed (1, One): " + removed);

// 24. replace(K key, V oldValue, V newValue)
map.put(1, "One");
boolean replaced = map.replace(1, "One", "New One");
System.out.println("Replaced (1, One -> New One): " + replaced);

// 25. replace(K key, V value)
map.replace(1, "One");
System.out.println("After replace: " + map);

// 26. comparator()
System.out.println("Comparator: " + map.comparator());

// 27. firstKey()
map.put(1, "One");
map.put(2, "Two");
System.out.println("First Key: " + map.firstKey());

// 28. lastKey()
System.out.println("Last Key: " + map.lastKey());

// 29. firstEntry()
System.out.println("First Entry: " + map.firstEntry());
```

```
        // 30. lastEntry()
        System.out.println("Last Entry: " + map.lastEntry());

        // 31. pollFirstEntry()
        System.out.println("Poll First Entry: " + map.pollFirstEntry());

        // 32. pollLastEntry()
        System.out.println("Poll Last Entry: " + map.pollLastEntry());

        // 33. descendingMap()
        System.out.println("Descending Map: " + map.descendingMap());

        // 34. descendingKeySet()
        System.out.println("Descending KeySet: " + map.descendingKeySet());

        // 35. navigableKeySet()
        System.out.println("Navigable KeySet: " + map.navigableKeySet());
    }
}
```

Batch 6: Concurrent Operations

```
import java.util.concurrent.ConcurrentSkipListMap;
import java.util.function.BiFunction;
import java.util.function.Function;

public class ConcurrentOperations {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
        map.put(1, "One");
        map.put(2, "Two");

        // 36. merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
remappingFunction)
        map.merge(1, "New One", (oldVal, newVal) -> oldVal + " " + newVal);
        System.out.println("After merge: " + map);
```

```
        // 37. compute(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction)
        map.compute(2, (k, v) -> v + " Updated");
        System.out.println("After compute: " + map);

        // 38. computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)
        map.computeIfAbsent(3, k -> "Three");
        System.out.println("After computeIfAbsent: " + map);

        // 39. computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction)
        map.computeIfPresent(3, (k, v) -> v + " Modified");
        System.out.println("After computeIfPresent: " + map);
    }
}
```

Batch 7: Bulk Operations and Miscellaneous

```
import java.util.HashMap;
import java.util.Map;
import java.util.concurrent.ConcurrentSkipListMap;

public class BulkOperations {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
        map.put(1, "One");
        map.put(2, "Two");

        // 40. putAll(Map<? extends K, ? extends V> m)
        Map<Integer, String> newMap = new HashMap<>();
        newMap.put(3, "Three");
        newMap.put(4, "Four");
        map.putAll(newMap);
        System.out.println("After putAll: " + map);

        // 41. clone()
```

```
        ConcurrentSkipListMap<Integer, String> clonedMap =
(ConcurrentSkipListMap<Integer, String>) map.clone();
        System.out.println("Cloned Map: " + clonedMap);

        // 42. equals(Object o)
        boolean isEqual = map.equals(clonedMap);
        System.out.println("Is equal: " + isEqual);

        // 43. hashCode()
        int hashCode = map.hashCode();
        System.out.println("HashCode: " + hashCode);

        // 44. toString()
        System.out.println("ToString: " + map.toString());
    }
}
```

Batch 8: ConcurrentMap Interface Methods

```
import java.util.concurrent.ConcurrentSkipListMap;
import java.util.function.BiConsumer;
import java.util.function.BiFunction;

public class ConcurrentMapMethods {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
        map.put(1, "One");
        map.put(2, "Two");

        // 45. getOrDefault(Object key, V defaultValue)
        String value = map.getOrDefault(3, "Default");
        System.out.println("GetOrDefault: " + value);

        // 46. forEach(BiConsumer<? super K, ? super V> action)
        map.forEach((k, v) -> System.out.println("Key: " + k + ", Value: " + v));

        // 47. replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
```

```
        map.replaceAll((k, v) -> v + " Updated");
        System.out.println("After replaceAll: " + map);
    }
}
```

Batch 9: NavigableMap Interface Methods

```
import java.util.Map;
import java.util.concurrent.ConcurrentSkipListMap;

public class NavigableMapMethods {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
        map.put(1, "One");
        map.put(2, "Two");
        map.put(3, "Three");

        // 48. lowerEntry(K key)
        Map.Entry<Integer, String> lowerEntry = map.lowerEntry(2);
        System.out.println("Lower Entry for 2: " + lowerEntry);

        // 49. lowerKey(K key)
        Integer lowerKey = map.lowerKey(2);
        System.out.println("Lower Key for 2: " + lowerKey);

        // 50. floorEntry(K key)
        Map.Entry<Integer, String> floorEntry = map.floorEntry(2);
        System.out.println("Floor Entry for 2: " + floorEntry);

        // 51. floorKey(K key)
        Integer floorKey = map.floorKey(2);
        System.out.println("Floor Key for 2: " + floorKey);

        // 52. ceilingEntry(K key)
        Map.Entry<Integer, String> ceilingEntry = map.ceilingEntry(2);
        System.out.println("Ceiling Entry for 2: " + ceilingEntry);
```

```
        // 53. ceilingKey(K key)
        Integer ceilingKey = map.ceilingKey(2);
        System.out.println("Ceiling Key for 2: " + ceilingKey);

        // 54. higherEntry(K key)
        Map.Entry<Integer, String> higherEntry = map.higherEntry(2);
        System.out.println("Higher Entry for 2: " + higherEntry);

        // 55. higherKey(K key)
        Integer higherKey = map.higherKey(2);
        System.out.println("Higher Key for 2: " + higherKey);
    }
}
```

Batch 11: Additional Methods

```
import java.util.concurrent.ConcurrentSkipListMap;
import java.util.function.BiFunction;
import java.util.function.Function;

public class AdditionalMethods {
    public static void main(String[] args) {
        ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();
        map.put(1, "One");
        map.put(2, "Two");

        // 59. forEach(BiConsumer<? super K, ? super V> action)
        map.forEach((k, v) -> System.out.println("Key: " + k + ", Value: " + v));

        // 60. replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
        map.replaceAll((k, v) -> v + " Updated");
        System.out.println("After replaceAll: " + map);

        // 61. merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
remappingFunction)
        map.merge(1, "New One", (oldVal, newVal) -> oldVal + " " + newVal);
        System.out.println("After merge: " + map);
```

```
        // 62. compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)
        map.compute(2, (k, v) -> v + " Updated");
        System.out.println("After compute: " + map);

        // 63. computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)
        map.computeIfAbsent(3, k -> "Three");
        System.out.println("After computeIfAbsent: " + map);

        // 64. computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)
        map.computeIfPresent(3, (k, v) -> v + " Modified");
        System.out.println("After computeIfPresent: " + map);
    }
}
```

What is an `EnumMap`?

An `EnumMap` is a specialized implementation of the `Map` interface in Java that is designed to work with **enum keys**. It is part of the `java.util` package and provides a highly efficient and compact way to store and retrieve values associated with enum constants.

Key Features of `EnumMap`:

1. **Enum Keys Only**: The keys in an `EnumMap` must be instances of a single enum type.
2. **High Performance**: Internally, `EnumMap` uses an array to store values, making it extremely fast for lookups, insertions, and deletions.
3. **Compact Storage**: Since the number of enum constants is fixed, `EnumMap` uses minimal memory compared to other `Map` implementations like `HashMap`.
4. **Natural Order**: By default, `EnumMap` maintains the natural order of enum constants (the order in which they are declared in the enum).
5. **Null Handling**: `EnumMap` allows `null` values but does not allow `null` keys.

How to Use `EnumMap`

Syntax EnumMap<EnumType, ValueType> map = new EnumMap<>(EnumType.class);

```java
import java.util.EnumMap;

enum Day {
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class EnumMapExample {
    public static void main(String[] args) {
        // Create an EnumMap with Day as the key type and String as the value type
        EnumMap<Day, String> schedule = new EnumMap<>(Day.class);

        // Add entries to the map
        schedule.put(Day.MONDAY, "Work");
        schedule.put(Day.TUESDAY, "Meeting");
        schedule.put(Day.WEDNESDAY, "Gym");

        // Retrieve values
        System.out.println("Schedule for Monday: " + schedule.get(Day.MONDAY));

        // Iterate over the map
        for (Day day : schedule.keySet()) {
            System.out.println(day + ": " + schedule.get(day));
        }
    }
}
```

Real-Life Use Cases of `EnumMap`

1. Scheduling Systems

- **Use Case**: Managing schedules or tasks for specific days of the week.
- **Example**:
-

```java
EnumMap<Day, String> dailyTasks = new EnumMap<>(Day.class);
dailyTasks.put(Day.MONDAY, "Team Meeting");
dailyTasks.put(Day.FRIDAY, "Project Deadline");
```

2. Configuration Management

- **Use Case**: Storing configuration settings for different environments (e.g., development, testing, production).
- **Example**:

```java
enum Environment {
```

```
    DEV, TEST, PROD
}

EnumMap<Environment, String> config = new EnumMap<>(Environment.class);
config.put(Environment.DEV, "Development Settings");
config.put(Environment.PROD, "Production Settings");
```

3. State Machines

- **Use Case**: Implementing state transitions in a finite state machine.
- **Example**:

```
enum State {
    IDLE, RUNNING, PAUSED, STOPPED
}

EnumMap<State, String> stateActions = new EnumMap<>(State.class);
stateActions.put(State.IDLE, "Waiting for input");
stateActions.put(State.RUNNING, "Processing data");
```

Advantages of `EnumMap`

1. **Performance**: Faster than `HashMap` for enum keys because it uses an array internally.
2. **Type Safety**: Ensures that only valid enum constants are used as keys.
3. **Compactness**: Uses less memory compared to other `Map` implementations.
4. **Order Preservation**: Maintains the natural order of enum constants.

When to Use `EnumMap`

- When the keys are enum constants.
- When performance and memory efficiency are critical.
- When you need to maintain the natural order of enum keys.

Comparison with `HashMap`

Feature	`EnumMap`	`HashMap`
Key Type	Only enum constants	Any object

Performance	Faster for enum keys	Slower for enum keys
Memory Usage	More compact	Less compact
Order Preservation	Maintains enum order	No guaranteed order
Null Keys	Not allowed	Allowed

Conclusion

`EnumMap` is a powerful and efficient data structure for scenarios where enum constants are used as keys. Its high performance, compact storage, and type safety make it ideal for use cases like scheduling systems, configuration management, state machines, and more. If you're working with enums, `EnumMap` is often the best choice for a `Map` implementation.

What is an Unmodifiable Map?

An **unmodifiable map** is a read-only view of a map in Java. It does not allow any modifications (such as adding, removing, or updating entries) after it is created. Any attempt to modify the map will result in an `UnsupportedOperationException`.
Unmodifiable maps are useful when you want to:

1. **Protect Data**: Ensure that the map's contents cannot be altered after creation.
2. **Share Data Safely**: Share a map with other parts of the program or external code without risking unintended modifications.
3. **Immutable Design**: Enforce immutability in your application design.

How to Create an Unmodifiable Map

Java provides several ways to create unmodifiable maps:

1. **Using `Collections.unmodifiableMap()`**:
   - Wraps an existing map and returns an unmodifiable view of it.
2. **Using `Map.of()` (Java 9+)**:
   - Creates an unmodifiable map with a fixed set of key-value pairs.
3. **Using `Map.copyOf()` (Java 10+)**:
   - Creates an unmodifiable copy of an existing map.

Program: Unmodifiable Map Example

```
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

public class UnmodifiableMapExample {
    public static void main(String[] args) {
        // Create a mutable map
        Map<String, Integer> mutableMap = new HashMap<>();
        mutableMap.put("Apple", 10);
        mutableMap.put("Banana", 5);
        mutableMap.put("Orange", 8);

        // 1. Create an unmodifiable map using Collections.unmodifiableMap()
        Map<String, Integer> unmodifiableMap =
Collections.unmodifiableMap(mutableMap);
        System.out.println("Unmodifiable Map: " + unmodifiableMap);

        // 2. Create an unmodifiable map using Map.of() (Java 9+)
        Map<String, Integer> unmodifiableMap2 = Map.of("Grapes", 15, "Mango", 20);
        System.out.println("Unmodifiable Map (Map.of()): " + unmodifiableMap2);

        // 3. Create an unmodifiable map using Map.copyOf() (Java 10+)
        Map<String, Integer> unmodifiableMap3 = Map.copyOf(mutableMap);
        System.out.println("Unmodifiable Map (Map.copyOf()): " + unmodifiableMap3);

        // Attempt to modify the unmodifiable maps
        try {
            unmodifiableMap.put("Pineapple", 7); // This will throw an exception
        } catch (UnsupportedOperationException e) {
            System.out.println("Cannot modify unmodifiableMap: " + e.getMessage());
        }

        try {
            unmodifiableMap2.put("Pineapple", 7); // This will throw an exception
        } catch (UnsupportedOperationException e) {
```

```
            System.out.println("Cannot modify unmodifiableMap2: " + e.getMessage());
        }

        try {
            unmodifiableMap3.put("Pineapple", 7); // This will throw an exception
        } catch (UnsupportedOperationException e) {
            System.out.println("Cannot modify unmodifiableMap3: " + e.getMessage());
        }

        // Modifying the original mutable map does not affect the unmodifiable map
        mutableMap.put("Pineapple", 7);
        System.out.println("Original Mutable Map after modification: " + mutableMap);
        System.out.println("Unmodifiable Map after original map modification: " +
unmodifiableMap);
    }
}
```

Explanation of the Program

1. Creating a Mutable Map

```
Map<String, Integer> mutableMap = new HashMap<>();
mutableMap.put("Apple", 10);
mutableMap.put("Banana", 5);
mutableMap.put("Orange", 8);
```

2. Creating an Unmodifiable Map Using `Collections.unmodifiableMap()`

```
Map<String, Integer> unmodifiableMap = Collections.unmodifiableMap(mutableMap);
System.out.println("Unmodifiable Map: " + unmodifiableMap);
```

- The `Collections.unmodifiableMap()` method wraps the mutable map and returns an unmodifiable view of it.
- Any attempt to modify this map will throw an `UnsupportedOperationException`.

---

3. Creating an Unmodifiable Map Using `Map.of()` (Java 9+)

Map<String, Integer> unmodifiableMap2 = Map.of("Grapes", 15, "Mango", 20);
System.out.println("Unmodifiable Map (Map.of()): " + unmodifiableMap2);

- The `Map.of()` method creates an unmodifiable map with a fixed set of key-value pairs.
- This method is convenient for creating small, immutable maps.

---

4. Creating an Unmodifiable Map Using `Map.copyOf()` (Java 10+)

Map<String, Integer> unmodifiableMap3 = Map.copyOf(mutableMap);
System.out.println("Unmodifiable Map (Map.copyOf()): " + unmodifiableMap3);

- The `Map.copyOf()` method creates an unmodifiable copy of the mutable map.
- This method ensures that the unmodifiable map is independent of the original map.

---

5. Attempting to Modify Unmodifiable Maps

```
try {
   unmodifiableMap.put("Pineapple", 7); // This will throw an exception
} catch (UnsupportedOperationException e) {
   System.out.println("Cannot modify unmodifiableMap: " + e.getMessage());
}
```

6. Modifying the Original Mutable Map

```
mutableMap.put("Pineapple", 7);
System.out.println("Original Mutable Map after modification: " + mutableMap);
System.out.println("Unmodifiable Map after original map modification: " + unmodifiableMap);
```

- Modifying the original mutable map does not affect the unmodifiable map created using `Collections.unmodifiableMap()`.
- However, the unmodifiable map created using `Map.copyOf()` is completely independent of the original map.

Output of the Program

```
Unmodifiable Map: {Apple=10, Banana=5, Orange=8}
Unmodifiable Map (Map.of()): {Grapes=15, Mango=20}
Unmodifiable Map (Map.copyOf()): {Apple=10, Banana=5, Orange=8}
Cannot modify unmodifiableMap: null
```

Cannot modify unmodifiableMap2: null
Cannot modify unmodifiableMap3: null
Original Mutable Map after modification: {Apple=10, Banana=5, Orange=8, Pineapple=7}
Unmodifiable Map after original map modification: {Apple=10, Banana=5, Orange=8}

Key Takeaways

1. **Unmodifiable maps are read-only**: They cannot be modified after creation.
2. **Different ways to create unmodifiable maps**:
    - `Collections.unmodifiableMap()`: Wraps an existing map.
    - `Map.of()`: Creates a small, fixed-size unmodifiable map.
    - `Map.copyOf()`: Creates an independent unmodifiable copy of a map.
3. **Use cases**:
    - Protecting data from unintended modifications.
    - Sharing data safely in a multi-threaded environment.
    - Enforcing immutability in your application design.