Complete Guide to HashMap in Java: From Basics to Advanced

This comprehensive guide covers **everything** you need to know about `HashMap` in Java, from its basic functionality to advanced concepts and interview-specific details. Whether you are a beginner or an expert, this will prepare you thoroughly.

📖 What is a HashMap?

A `HashMap` is a Java class that implements the `Map` interface and allows you to store data as **key-value pairs**. It provides **fast access** to elements using hashing.

Key Characteristics

- 🔑 **Key**: Unique identifier for a value.
- 📦 **Value**: Data associated with the key.
- ✅ **Allows nulls**: One `null` key and multiple `null` values.
- 🔀 **Unordered**: Does not maintain insertion order.
- ⚡ **Efficient**: O(1) time complexity (average case) for `put()` and `get()` operations.
- 🔒 **Not thread-safe**: Use `ConcurrentHashMap` for multithreaded environments.

🌟 Features of HashMap

1. **Hashing**: Keys are hashed into bucket indices for fast lookup.
2. **Collision Handling**: Uses linked lists or balanced trees for collisions.
3. **Resizing**: Automatically resizes when the load factor is exceeded.
4. **Customizable Capacity and Load Factor**: You can configure them for performance optimization.

🛠️ How HashMap Works Internally

1. Hashing

The **key's `hashCode()`** determines the bucket index using:
index = (hashCode & (capacity - 1))

-

2. Buckets

- The hash table is an array of buckets.
- Each bucket can store multiple entries if collisions occur.

3. Collision Handling

- **Before Java 8**: Linked lists are used within buckets.
- **Java 8 and later**: Converts linked lists to balanced trees for faster lookups if a bucket has more than 8 entries.

4. Resizing

- When the number of elements exceeds the threshold (capacity × load factor), `HashMap` resizes:
    1. Doubles the array size.
    2. Rehashes all existing entries into the new table.

🌟 **Visualizing HashMap's Internal Structure**

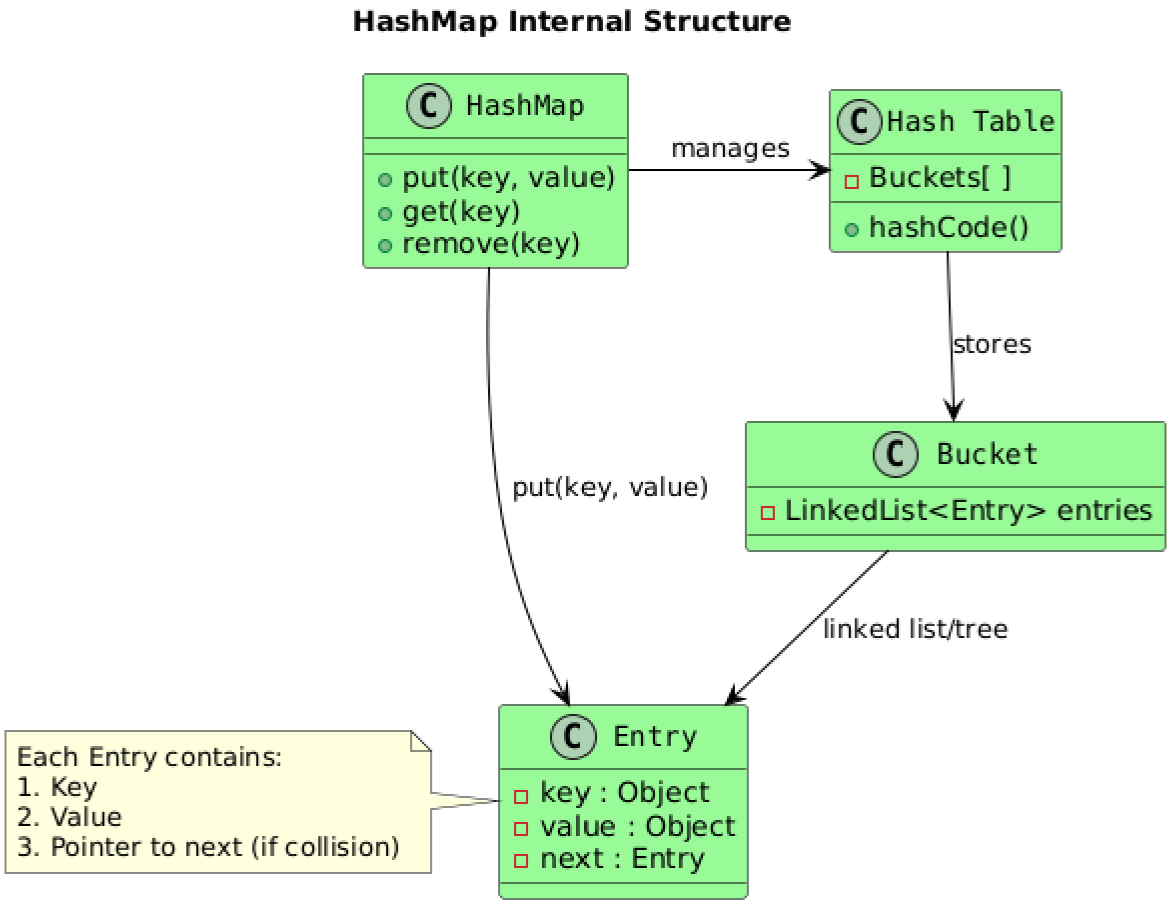

Diagram Explanation

- **HashMap**: Manages the hash table and provides APIs like `put()`, `get()`, and `remove()`.
- **Hash Table**: The underlying array of buckets where data is stored.
- **Bucket**: Each bucket contains a linked list or balanced tree (depending on Java version and collision frequency).
- **Entry**: Represents each key-value pair in the `HashMap`, with a pointer to the next entry in case of collisions.

🛠️ Commonly Used Methods in HashMap

🛠️ HashMap Methods

Here's a list of commonly used methods in `HashMap`:

Method	Description
`put(K key, V value)`	Adds or updates a key-value pair.
`get(Object key)`	Retrieves the value associated with the given key.
`remove(Object key)`	Removes the key-value pair associated with the given key.
`containsKey(Object key)`	Checks if the specified key exists in the map.
`containsValue(Object value)`	Checks if the specified value exists in the map.
`size()`	Returns the number of key-value pairs in the map.
`isEmpty()`	Checks if the map is empty.
`keySet()`	Returns a `Set` of all keys.
`values()`	Returns a `Collection` of all values.
`entrySet()`	Returns a `Set` of all key-value pairs (as `Map.Entry` objects).
`putIfAbsent(K key, V value)`	Adds a key-value pair only if the key is not already associated with a value.
`replace(K key, V value)`	Replaces the value for a key if it exists.
`clear()`	Removes all key-value pairs from the map.
`forEach(BiConsumer action)`	Performs the given action for each key-value pair in the map.

✍️ Examples

Example 1: Basic Usage

```
import java.util.HashMap;

public class Main {

    public static void main(String[] args) {

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

        map.put("Alice", "123-456");
```

```
        map.put("Bob", "987-654");

        System.out.println(map.get("Alice")); // Output: 123-456

    }

}
```

Example 2: Word Counter

Count occurrences of words in a string:

```
String text = "Java is fun and Java is powerful";

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

for (String word : text.split(" ")) {

    wordCount.put(word, wordCount.getOrDefault(word, 0) + 1);

}

System.out.println(wordCount); // Output: {Java=2, is=2, fun=1, and=1, powerful=1}
```

🔄 Iteration Methods

1. Key Set Iteration

```
for (String key : map.keySet()) {

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

}
```

2. Entry Set Iteration (Preferred)

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

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

}

3. Java 8 forEach

map.forEach((key, value) -> System.out.println(key + ": " + value));

🔒 Thread Safety in HashMap

`HashMap` is **not thread-safe**. For multithreading, use:

Synchronized HashMap:

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

1.
2. **ConcurrentHashMap**:
   - Divides the hash table into segments for thread-safe concurrent access.
   - Does not allow `null` keys or values.

⚠️ Common Pitfalls

1. **Using Mutable Keys**:
   - If a key's state changes after insertion, it becomes unretrievable.
   - Always use immutable objects like `String` as keys.
2. **Improper `hashCode()` and `equals()`**:
   - Ensure `hashCode()` and `equals()` are consistent for custom keys.
3. **High Load Factors**:
   - A high load factor (e.g., >0.75) increases collision chances and reduces performance.

🎯 Real-World Use Cases

1. **Caching**:
   - Store frequently accessed data for fast retrieval.
2. **Indexing**:
   - Use for indexing data in databases or search engines.
3. **Grouping Data**:
   - Group data based on a common key, like organizing employees by department.

🎓 Interview-Specific Questions

Beginner-Level

1. **What is a HashMap?**
   - A data structure that stores key-value pairs using hashing.
2. **What is the default capacity and load factor of HashMap?**
   - Default capacity: **16**, default load factor: **0.75**.

Intermediate-Level

3. **What happens if two keys have the same hash code?**
   - A collision occurs. Colliding entries are stored in the same bucket, using a linked list or balanced tree.
4. **How does HashMap resize?**
   - HashMap resizes when the size exceeds capacity × load factor, doubling its capacity and redistributing entries.

Advanced-Level

5. **Why is HashMap's capacity always a power of 2?**
   - Ensures efficient bucket calculation using bitwise operations.
6. **What is the difference between HashMap and ConcurrentHashMap?**
   - `HashMap` is not thread-safe, while `ConcurrentHashMap` is thread-safe and optimized for concurrency.

🏅 Coding Challenges

Challenge 1: Find the First Non-Repeating Character

```
public class NonRepeating {

    public static void main(String[] args) {

        String str = "swiss";

        HashMap<Character, Integer> charCount = new HashMap<>();


        for (char c : str.toCharArray()) {

            charCount.put(c, charCount.getOrDefault(c, 0) + 1);

        }


        for (char c : str.toCharArray()) {

            if (charCount.get(c) == 1) {

                System.out.println("First non-repeating character: " + c);

                break;

            }

        }

    }

}
```

Challenge 2: Group Anagrams

```
import java.util.*;

public class GroupAnagrams {

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

```
        String[] words = {"bat", "tab", "cat", "act", "dog"};
        HashMap<String, List<String>> anagramGroups = new HashMap<>();
        for (String word : words) {
            char[] chars = word.toCharArray();
            Arrays.sort(chars);
            String sorted = new String(chars);
            anagramGroups.putIfAbsent(sorted, new ArrayList<>());
            anagramGroups.get(sorted).add(word);
        }
        System.out.println(anagramGroups.values());
    }
}
```

Output:

```
[[bat, tab], [cat, act], [dog]]
```

🔍 Additional Insights

1. Load Factor Tuning

- The default load factor (0.75) provides a good trade-off between space and time complexity.
- **When to adjust?**
  - If memory is limited and read operations dominate, a **higher load factor** reduces space usage but increases collision likelihood.
  - If fast access is crucial, a **lower load factor** minimizes collisions at the cost of higher memory usage.

---

2. Comparison with Other Data Structures

Feature	HashMap	TreeMap	LinkedHashMap
Order	Unordered	Sorted (natural or custom)	Insertion order preserved
Performance	O(1) for get/put	O(log n) for get/put	O(1) for get/put
Use Case	Fast lookups	Sorted data access	Ordered iteration

---

3. Custom Key Class for HashMap

If you use a custom object as a key in a `HashMap`, you **must override** `hashCode()` and `equals()`. Without this, the `HashMap` will not work correctly.

Example:

```
class Employee {
    int id;
    String name;

    Employee(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public int hashCode() {
        return id; // Use ID as a unique hash
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Employee other = (Employee) obj;
        return id == other.id;
    }
}

public class Main {
    public static void main(String[] args) {
        HashMap<Employee, String> map = new HashMap<>();
        map.put(new Employee(1, "Alice"), "Developer");
```

```
        map.put(new Employee(2, "Bob"), "Manager");

        System.out.println(map.get(new Employee(1, "Alice"))); // Output: Developer
    }
}
```

---

4. HashMap Performance Optimization

1. **Avoid Poorly Distributed HashCodes**:
   - A poorly implemented `hashCode()` can lead to excessive collisions.
   - Use prime numbers in hash code calculations for better distribution.
2. **Minimize Resizing**:
   - Initialize the `HashMap` with an appropriate size if you know the approximate number of elements.

---

5. Debugging HashMap Issues

- **Common Issues**:
  - Missing entries due to incorrect `hashCode()` or `equals()` implementation.
  - Performance degradation caused by excessive collisions.
- **Tools**:
  - Use Java Profiler (e.g., JVisualVM) to monitor bucket usage and resizing behavior.
  - Log `hashCode()` values and bucket indices to diagnose collision problems.

🎓 Tips for Interviews

1. Understand When to Use HashMap

- Use a `HashMap` when:
  - You need **constant time performance** for lookups and inserts.
  - **Order of elements doesn't matter**.

2. Explain the Evolution of Collision Resolution

- Be ready to explain how `HashMap` evolved from linked lists (Java 7) to balanced trees (Java 8+) to improve performance.

3. Real-World Use Case Examples

- Be prepared to discuss scenarios like:
  - **Caching in web applications**.
  - **Indexing in databases**.
  - **Grouping data** (e.g., anagram grouping, word frequency counting).

🛠️ Practice Coding Challenges

1. Find All Duplicates in an Array

```
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

public class FindDuplicates {
    public static List<Integer> findDuplicates(int[] nums) {
        HashMap<Integer, Integer> map = new HashMap<>();
        List<Integer> duplicates = new ArrayList<>();

        for (int num : nums) {
            map.put(num, map.getOrDefault(num, 0) + 1);
        }

        for (int key : map.keySet()) {
            if (map.get(key) > 1) {
                duplicates.add(key);
            }
        }

        return duplicates;
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 3, 1, 2, 4};
        System.out.println(findDuplicates(nums)); // Output: [1, 2]
    }
}
```

---

2. Top K Frequent Elements

```java
import java.util.*;

public class TopKFrequent {
    public static List<Integer> topKFrequent(int[] nums, int k) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for (int num : nums) {
            map.put(num, map.getOrDefault(num, 0) + 1);
        }

        PriorityQueue<Map.Entry<Integer, Integer>> pq = new PriorityQueue<>(
            (a, b) -> b.getValue() - a.getValue()
        );

        pq.addAll(map.entrySet());

        List<Integer> result = new ArrayList<>();
        while (k-- > 0) {
            result.add(pq.poll().getKey());
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        System.out.println(topKFrequent(nums, k)); // Output: [1, 2]
    }
}
```

🔑 Key Takeaways

- **Understand Internals**: Explain hashing, bucket mechanics, collision handling, and resizing.
- **Apply Best Practices**: Use immutable keys, tune capacity/load factor, and ensure proper `hashCode()` and `equals()` implementations.
- **Showcase Problem Solving**: Discuss real-world use cases and demonstrate coding proficiency with practical challenges.
