Internal working of HashMap in Java 8 & Beyond

By WebRewrite | May 3, 2025 | Java

How HashMap internally works in java is the most asked interview question. Even with just 1–2 years of experience, you've likely used HashMap in your code. However, many developers are unaware of how it actually works under the hood.

This article dives deep into the internal mechanics of HashMap, explaining how the get, put, operations work behind the scenes. I am going to discuss the internal working of HashMap in java with diagrams and code examples.

How HashMap works internally video tutorial

What Really Happens When You Instantiate a HashMap in Java

When you create an instance of HashMap –

```
1 Map<String, Integer> productCountMap = new HashMap<>();
```

Java doesn't immediately allocate the internal array. Instead, it sets up a few default values:

Capacity = 16
Load Factor = 0.75
Threshold = 12 (i.e., capacity × load factor)
Table = null (the actual array is not yet created)

This is called **lazy initialization** — it helps conserve memory until the first element is added.

HashMap First put() Operation Explained Step-by-Step

Let's say you perform your first insertion like this:

```
1 map.put("book", 20);
2
```

Here's what happens step-by-step:

1. **Array Initialization:**
Since this is the first put() call, Java hasn't yet created the internal array (table). So it initializes it with a default size of 16.

2. **Hash Code Calculation:**
The hash code of the key "book" is calculated using the hashCode() method.
Let's assume:

```
1 "book".hashCode() = 93029210
2
```

3. **Hash Transformation:**
This raw hash code is further processed using bit manipulation (like XOR and shifts) to spread the bits more evenly. This helps reduce collisions.

4. **Index Calculation:**
The final index (bucket) is determined using:

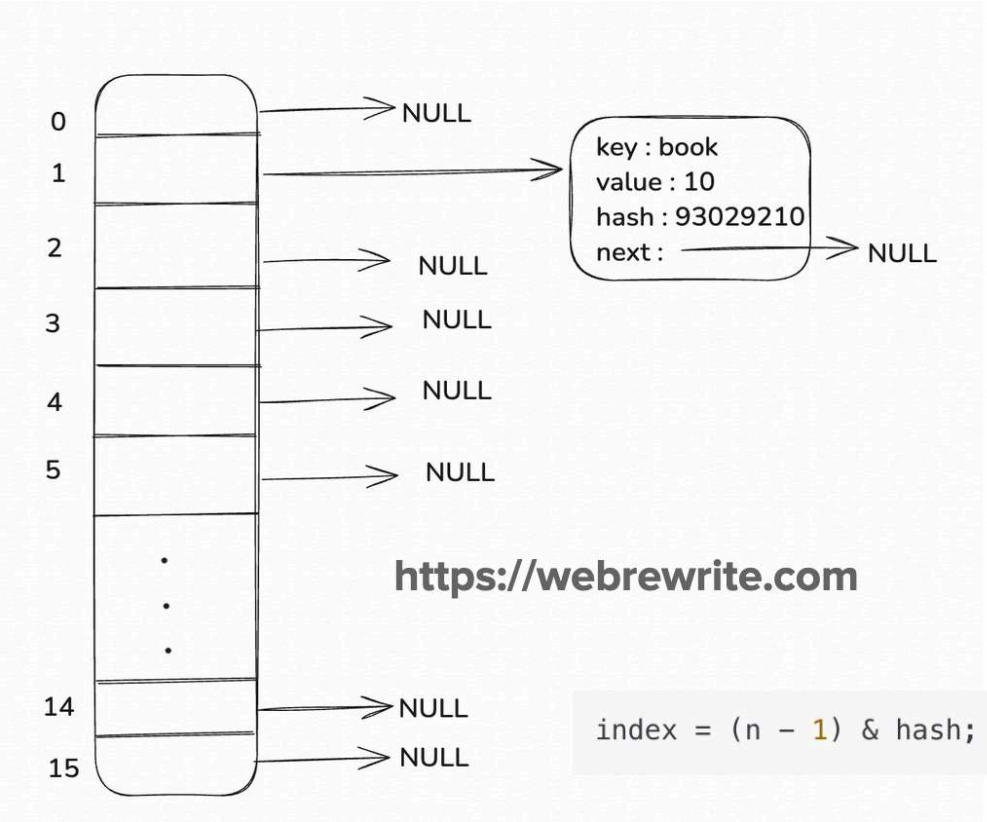
```
1 index = (n - 1) & hash
2 index = (16 - 1) & 93029210 = 15 & 93029210 = 1
```

5. **Node Insertion:**
At index 1 of the array, Java inserts a new node.
Each node in the HashMap holds the following:

key → "book"
value → 20

hash → transformed hash of the key

next → reference to the next node (for handling collisions via LinkedList)



Let's do few more put operation —

productCount.put("pen",20);

When you add more entries, each of these goes through the same process –

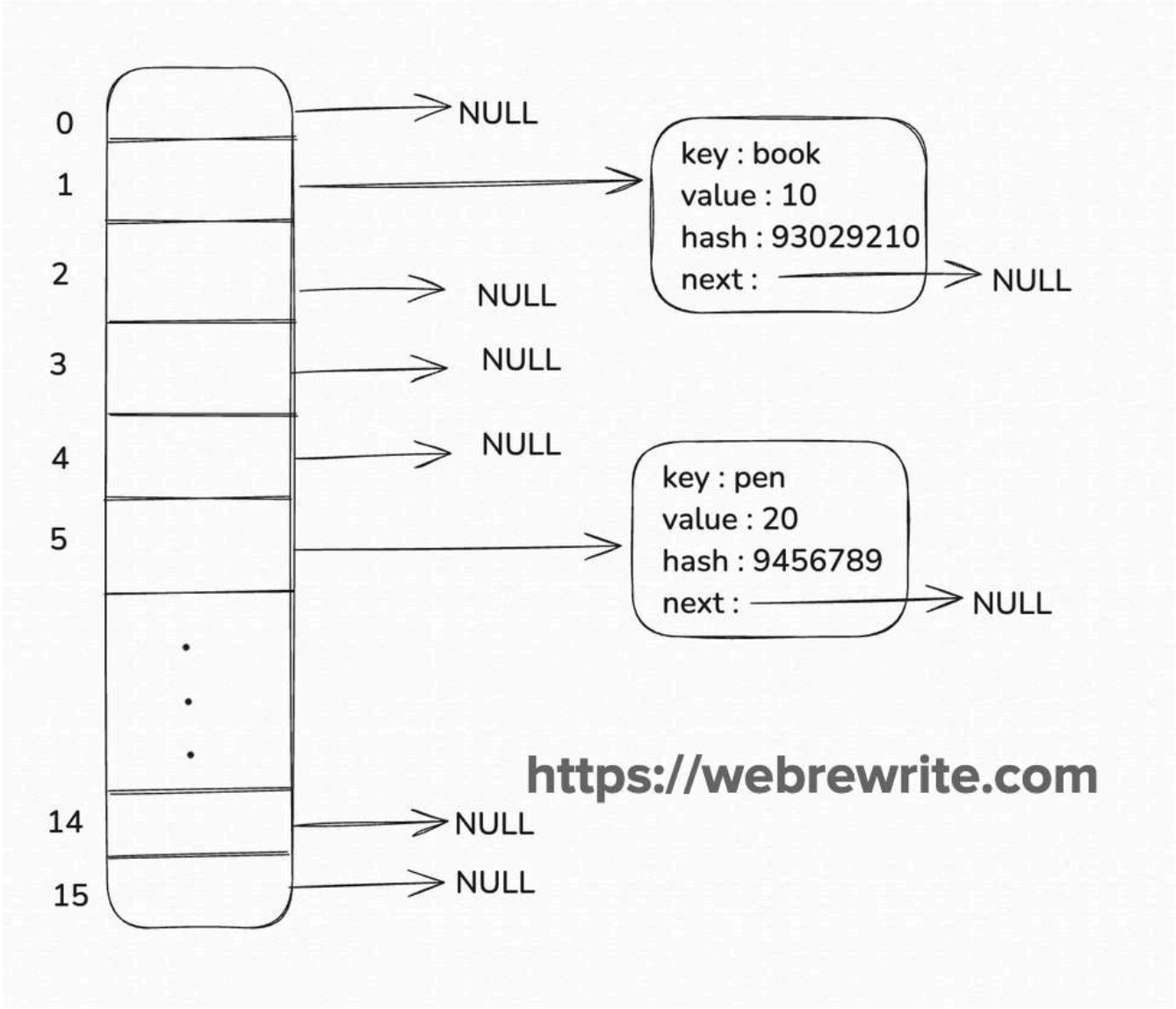
1. Compute hash code of the key.

```
1 hashcode("pen")= 9456789<br>
```

2. Calculate the index.

```
1 index = (16-1) & 9456789
2 index = 5
```

3. Insert the node at the computed index.



How HashMap Handles Collision

Initially, entries at the same index are stored in a **Linked List.**
If more than 8 entries collide at the same index (and the array size is >= 64), Java switches to a balanced tree for faster access (O(logn)).

Let's assume we are executing this put operation –

productCount.put("pencil", 80);

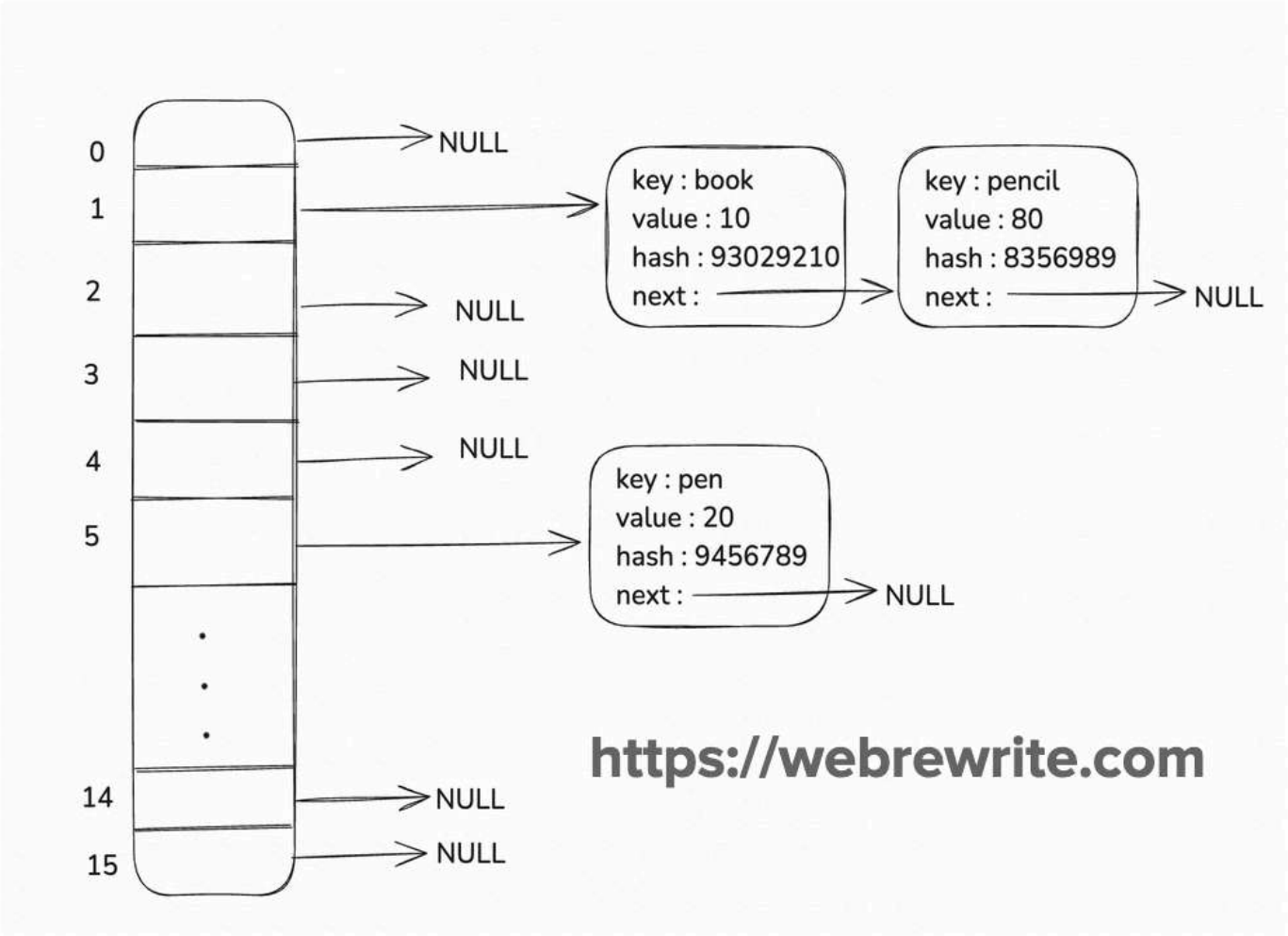
1. First step is to compute hash code

```
1 hashcode("pencil")= 8356989
```

2. Calculate index

```
1 Index = (16-1) & 8356989
2 Index = 1
```

3. Insert the node at the computed index. Since index 1 already contains a node, it's a **collision**. Java handles this by **appending the new node to the end of the linked list at that index.**



How HashMap Resizing Works (Behind the Scenes)

1. With every `put()` operation, Java checks whether the current number of entries exceeds the **threshold**.

By default:

Initial Capacity = 16
Load Factor = 0.75
So, **Threshold** = 16 × 0.75 = **12**

2. When the **13th entry** is added, **resizing is triggered**.

3. The internal array's size is **doubled** (from 16 to 32).

4. All existing entries are **rehashed and redistributed** into the new array since the index calculation depends on the updated array size.

How HashMap get() Method Works

When we call HashMap get() method like this

```
1 productCount.get(“book).
2
```

Here, what happens internally –

Compute the hash code of the key.
Calculate the index
Go to the bucket.

In bucket –

If no node is found then return null.
If node is found:

Then it checks –

* check if **node key equals to key**

* If yes return value.

* If not, traverse the list/tree until a match is found.

* If match found return the key else return null.

Conclusion

HashMap leverages hashing to enable fast data retrieval.

It is backed by an **array of buckets** for storing entries.

Collisions are handled using Linked Lists or balanced Trees (like Red-Black Trees).

It **automatically resizes** when the number of entries exceeds the load threshold.

By default, **HashMap is not thread-safe**, so external synchronization is needed for concurrent access.

Related Posts:

1. **Java Program to Reverse a String using Stack**
2. **Find First and Last Position of Element in Sorted Array**
3. **Find Common Elements in Three Sorted Arrays – Java Code**
4. **Find Duplicate Characters in a String | Java Code**

Tagged HashMap, How HashMap Internally Works, Interview Questions, Java. Bookmark the permalink.

About WebRewrite

I am technology lover who loves to keep updated with latest technology. My interest field is Web Development.

View all posts by WebRewrite →

« Blocking Queue Implementation in Java

Load Comments

Support Me on PayPal

Programming Video Tutorials

Search

Related Posts (YARPP)

- **Race Condition in Java: Causes, Examples, and Solutions**
- **Find Second Largest Number in Array**
- **Check Balanced Parentheses in an Expression**
- **Sum of Digits of a Number using Recursion – Java Code**

Topics

Multithreading	Binary Search
Linked List	Programming
Binary Tree	Sliding Window
Hash Map	Composer
Java	Memcache

Queue Javascript

Stack Solid Principles