HashMap Internal Working in Java

■ Simplified Overview: How HashMap Works

Imagine a library where instead of searching through every shelf for a book, a librarian uses a special formula based on the book's title to instantly know which specific shelf it's on. A HashMap works in a similar way.

• **Storage and the JVM:** When you create a HashMap with **new HashMap<>()**, the JVM allocates space for it in the **heap memory**.

• **The Bucket Array:** Inside the HashMap is an array of "buckets" (like shelves in the library). By default, this array has 16 buckets.

• **Storing a Key-Value Pair (The put method):**
1. **Hashing the Key:** When you add a pair (map.put("Tony", "Iron Man")), the HashMap asks the key ("Tony") for its hash code.
2. **Finding the Bucket:** It calculates an index using **HashCode(Key) & (n - 1)**.
3. **Handling Collisions:** If two keys land in the same bucket, a linked list or tree is used.

• **Retrieving a Value (The get method):** The HashMap again hashes the key to find the correct bucket and then checks with **equals()** method.

■■■ Professional Deep Dive: Architecture, Memory, and Mechanics

Internal Architecture on the Heap:
The HashMap object on the heap contains a reference to **Node[] table** array. Each Node stores:
• final int hash
• final K key
• V value
• Node next

Capacity, Load Factor, and Rehashing:
• Initial Capacity: Default 16
• Load Factor: Default 0.75
• Rehashing: Doubles size when threshold exceeded

Hashing Process in HashMap:

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

Collision Resolution (Java 8+):
• TREEIFY_THRESHOLD: Bucket converts to Red-Black Tree if > 8 nodes.
• UNTREEIFY_THRESHOLD: Tree converts back to LinkedList if < 6 nodes.

■ Key Characteristics Summary

Aspect	Description
JVM Storage	HashMap and Node[] stored in heap
Null Keys/Values	Allows one null key and multiple null values
Duplicates	Not allowed, overwrites existing value
Time Complexity	O(1) average for put/get
Synchronization	Not synchronized, use ConcurrentHashMap

■ Project Code Example: UPI-like Transaction Cache

```
import java.util.HashMap;

public class UPITransactionCache {
    private HashMap transactionCache;

    public UPITransactionCache() {
        this.transactionCache = new HashMap<>(100);
    }

    public void cacheTransaction(Transaction transaction) {
        transactionCache.put(transaction.getTransactionId(), transaction)
;
    }

    public Transaction getTransactionStatus(String transactionId) {
        return transactionCache.get(transactionId);
    }

    public static void main(String[] args) {
        UPITransactionCache cache = new UPITransactionCache();
        Transaction tx1 = new Transaction("TXN001", "UPI123", "UPI456", 1
50.75);
        cache.cacheTransaction(tx1);
        Transaction retrievedTx = cache.getTransactionStatus("TXN001");
        if (retrievedTx != null) {
            System.out.println("Transaction Amount: " + retrievedTx.getAm
ount());
        }
    }
}

class Transaction {
    private String transactionId;
    private String fromVPA;
    private String toVPA;
    private double amount;

    public Transaction(String transactionId, String fromVPA, String toVPA
, double amount) {
        this.transactionId = transactionId;
        this.fromVPA = fromVPA;
        this.toVPA = toVPA;
        this.amount = amount;
    }
    public String getTransactionId() { return transactionId; }
    public double getAmount() { return amount; }
}
```