A **HashMap** in Java is part of the java.util package and implements the Map interface.

**Hash Table:** Internally, a HashMap uses an array of **buckets** (or slots) to store key-value pairs. Each bucket is typically implemented as a linked list or a tree (depending on the size and the number of collisions).

**Hash Function:** A key is passed through a hash function, which computes an integer value (hash code). This value is used to determine the index in the array (bucket) where the key-value pair should be stored.

**Collision Handling:** When multiple keys hash to the same bucket (i.e., a collision occurs), HashMap handles it in two ways:

* **Chaining:** Initially, collisions are handled by creating a linked list of entries in the same bucket.
* **Treeification (JDK 8 and later):** If a bucket becomes too large (e.g., more than 8 entries), the linked list is converted into a **balanced tree** (Red-Black Tree) for faster lookups.

**Put:** When a key-value pair is inserted, the key is hashed, and the index of the array (bucket) is computed. If no entry already exists in that bucket, the pair is inserted. If an entry with the same key exists, the value is updated.

**Get:** To retrieve a value by its key, the key is hashed again, and the corresponding bucket is accessed. The linked list or tree is then traversed to find the entry with the matching key.

**Remove:** To remove a key-value pair, the key is hashed, the correct bucket is found, and the entry is removed from the list or tree.

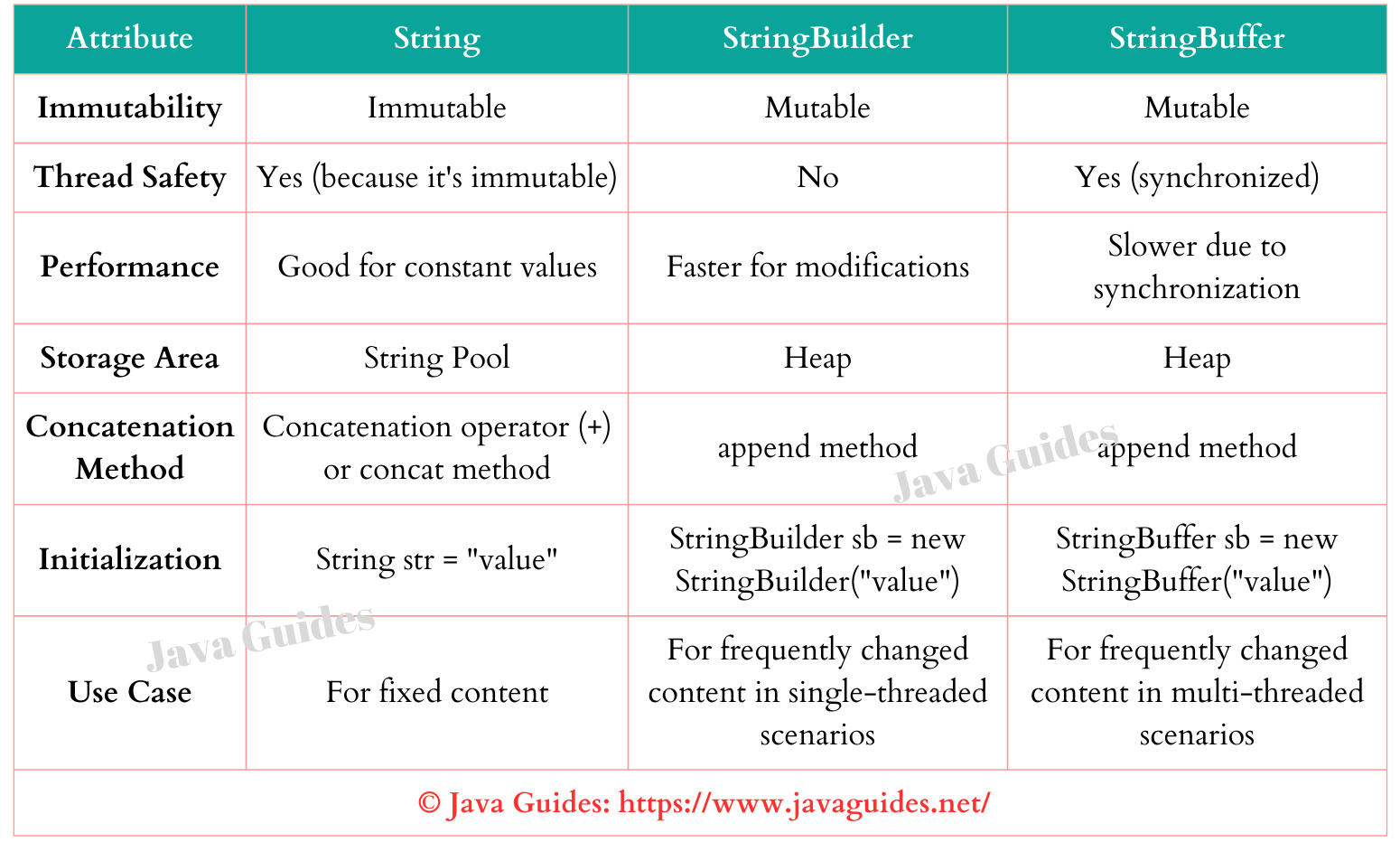
**Initial Capacity:** The initial size of the array (or table) is set when the HashMap is created. The default is typically 16.

**Load Factor:** The load factor determines when the HashMap will resize. The default load factor is 0.75, meaning the table will resize when the number of entries exceeds 75% of the current capacity.

**Resizing:** When the HashMap exceeds the threshold (capacity × load factor), it automatically increases the capacity, typically doubling the size of the array. When this happens, all existing entries are rehashed and redistributed into the new array.

* **hashCode():** The hashCode() method of the key is used to compute the index. The hashCode() should be well-distributed to minimize collisions and avoid performance degradation.
* **equals():** Once the correct bucket is located, the HashMap uses the equals() method of the key to determine if the key matches the one in the bucket. This is especially important in case of hash collisions.
* **Average Case (O(1)):** In an ideal situation (no collisions), the time complexity for put(), get(), and remove() operations is O(1), as accessing the correct bucket and finding the entry requires constant time.
* **Worst Case (O(n)):** If all keys hash to the same bucket (causing a long linked list or tree), the time complexity can degrade to O(n), where n is the number of entries in the HashMap. However, with treeification, the worst-case time complexity is O(log n), thanks to the balanced tree structure.

A HashMap is **not thread-safe**. If multiple threads access and modify the map concurrently, it may lead to data inconsistency. To make it thread-safe, you can either:

* Use Collections.synchronizedMap() to wrap the HashMap.
* Use ConcurrentHashMap for thread-safe, concurrent access with higher scalability.
*   
    
    
    
  HashSet<E> is backed by a HashMap<E, Object>.
* It uses the elements you add as **keys** in the underlying HashMap.
* A constant dummy object is used as the **value** for all keys.

When you add an element to a HashSet:

* The element is passed to the put() method of the internal HashMap as a key.
* The value is always the dummy object (PRESENT).

Internally, HashMap uses the **hashCode()** and **equals()** methods:

* hashCode() determines the bucket location.
* equals() is used to check if the element already exists (to prevent duplicates).

If two elements have the same hash code, HashMap uses equals() to resolve collisions.

