**🧠 Basic Idea**

| **Feature** | **SynchronizedMap** | **ConcurrentHashMap** |
| --- | --- | --- |
| Thread-safe? | ✅ Yes (via full lock) | ✅ Yes (via fine-grained locking) |
| Performance | 🚫 Slower (locks whole map) | ⚡ Faster (locks portions only) |
| Null keys/values allowed? | ✅ 1 null key, many null values | ❌ Not allowed |
| Introduced in | JDK 1.2 (Collections.synchronizedMap) | JDK 1.5 |

**🔁 1. SynchronizedMap — the old way**

Wraps a normal map with a **synchronized wrapper**.

**✅ Example:**

java

CopyEdit

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

Internally, it synchronizes every method:

java

CopyEdit

public synchronized V get(Object key) {

return map.get(key);

}

**❌ Problem:**

* Only **one thread** can access the map at a time.
* Causes **lock contention** — bad for performance under high concurrency.

**⚙️ 2. ConcurrentHashMap — the modern way**

**✅ Example:**

java

CopyEdit

Map<String, String> map = new ConcurrentHashMap<>();

**✅ Internal working:**

* Splits the map into **segments (buckets)**.
* Uses **fine-grained locking** — only the **specific segment or bucket is locked**, not the entire map.
* Allows **multiple threads to read/write different parts simultaneously**.

**🔥 Java 8 and above:**

* Uses **CAS (Compare-And-Swap)** for updates.
* Lock-free reads (most operations don't even block).

**🚫 Null Support**

| **Operation** | **HashMap** | **SynchronizedMap** | **ConcurrentHashMap** |
| --- | --- | --- | --- |
| put(null, value) | ✅ | ✅ | ❌ throws NPE |
| put(key, null) | ✅ | ✅ | ❌ throws NPE |

ConcurrentHashMap avoids null to prevent ambiguity with get() returning null.

**✅ When to Use What?**

| **Scenario** | **Recommendation** |
| --- | --- |
| Single-threaded | HashMap |
| Multi-threaded (low concurrency) | SynchronizedMap |
| Multi-threaded (high concurrency) | ConcurrentHashMap ✅ |
| Need predictable iteration order | Consider ConcurrentSkipListMap or Collections.synchronizedMap(new LinkedHashMap<>()) |

**💡 Summary**

| **Feature** | **SynchronizedMap** | **ConcurrentHashMap** |
| --- | --- | --- |
| Synchronization | Entire Map (coarse-grained) | Bucket/Segment (fine-grained) |
| Performance | Slower | Faster under concurrency |
| Null keys/values | ✅ Yes | ❌ No |
| Thread Safety | ✅ | ✅ |

Let me know if you want a performance test example or code comparison!

Awesome follow-up! 🔍 Let's dive into what **CAS (Compare-And-Swap)** and **lock-free reads** mean in the context of **Java 8+ ConcurrentHashMap** — this is where the *magic* happens for high-performance concurrency. 🧙‍♂️

**🔥 What is CAS (Compare-And-Swap)?**

**CAS** is a low-level, atomic operation used for **thread-safe updates without locking**.

**⚙️ The idea:**

“I’ll update a value **only if** it hasn't changed since I last saw it.”

**Pseudocode:**

if (currentValue == expectedValue) {

set newValue

}

**✅ In Java:**

CAS is implemented via **Unsafe** class or **VarHandle** (in Java 9+) and used under the hood by many concurrent collections and Atomic\* classes.

**🚀 Why CAS is fast:**

* **No need to acquire a lock**.
* Uses CPU-level atomic instructions (compareAndSwapInt, etc).
* Much lower overhead than synchronized.

**🔍 In ConcurrentHashMap (Java 8+):**

**✅ How it uses CAS:**

* When adding or updating a key-value pair, it uses **CAS** to check if the slot is empty or still contains the expected value.
* If another thread modified the map before your thread finishes — your CAS fails and retries.

This avoids heavy locks like synchronized or ReentrantLock for most operations.

**🔓 Lock-free reads:**

In Java 8+:

java

CopyEdit

map.get("someKey");

* **Does not acquire any lock**.
* Just reads the volatile value from the memory.
* Thanks to **volatile** and **memory visibility guarantees**, the read is thread-safe.

So, multiple threads can read **at the same time** without waiting 🔥

**🧠 Visualization**

java

CopyEdit

Thread 1: Reads the value of key "A" (no lock)

Thread 2: Tries to update value of key "A"

→ Uses CAS to ensure no one changed it

→ If value still same → update

→ If value changed → retry

**✅ Benefits of CAS in ConcurrentHashMap**

| **Feature** | **With Lock** | **With CAS** |
| --- | --- | --- |
| Performance | 🚫 Slower under load | ⚡ Very fast |
| Blocking | ✅ Yes | ❌ No (mostly) |
| Thread contention | High | Low |
| Deadlocks | Possible | ❌ Never |

**🧪 Bonus: CAS is used in**

* AtomicInteger, AtomicReference
* ConcurrentHashMap
* StampedLock (optimistic locking)
* LongAdder, CopyOnWriteArrayList