**✅ Compare-And-Swap (CAS) in Java – Explained for Interviews**

**📌 What is Compare-And-Swap (CAS)?**

**Compare-And-Swap (CAS)** is an **atomic** operation used in **concurrent programming** to update a shared variable **without using locks**. It is a fundamental concept behind **non-blocking algorithms** in Java.

It works by:

1. **Comparing** the current value of a variable with an expected value.
2. **Swapping** (updating) the value if it matches the expected value.
3. **Retrying** if the value has changed by another thread.

**💡 Why use CAS?**  
It allows **multiple threads** to update a shared variable **concurrently** while maintaining **thread safety** without the overhead of **locking**.

**📌 How CAS Works – Step-by-Step**

1. **Read** the current value of a shared variable.
2. **Compare** it with the expected value.
3. If **matched**, **update** it to a new value.
4. If **not matched**, **retry** the operation.

**✅ CAS Method Signature (Atomic Classes in Java)**

boolean compareAndSet(expectedValue, newValue);

* **If expectedValue == currentValue** → Update to newValue and return true.
* **If expectedValue != currentValue** → No update, return false.

**📌 CAS Example Using AtomicInteger**

import java.util.concurrent.atomic.AtomicInteger;

public class CASExample {

public static void main(String[] args) {

AtomicInteger atomicValue = new AtomicInteger(0);

// Thread 1 - Incrementing the value

Thread t1 = new Thread(() -> {

int oldValue = atomicValue.get();

boolean updated = atomicValue.compareAndSet(oldValue, oldValue + 1);

System.out.println(Thread.currentThread().getName() + " Updated: " + updated + " -> " + atomicValue.get());

});

// Thread 2 - Incrementing the value

Thread t2 = new Thread(() -> {

int oldValue = atomicValue.get();

boolean updated = atomicValue.compareAndSet(oldValue, oldValue + 1);

System.out.println(Thread.currentThread().getName() + " Updated: " + updated + " -> " + atomicValue.get());

});

t1.start();

t2.start();

}

}

**✅ Output (Non-Deterministic due to concurrency)**

Thread-0 Updated: true -> 1

Thread-1 Updated: true -> 2

**📌 CAS vs. Synchronized (Lock-Free vs. Lock-Based)**

| **Feature** | **CAS (Compare-And-Swap)** | **synchronized (Lock-Based)** |
| --- | --- | --- |
| **Mechanism** | Atomic operation on memory | Blocking using object-level monitor |
| **Performance** | Faster (No blocking) | Slower (Thread contention, blocking) |
| **Scalability** | High (Efficient for many threads) | Low (Contention increases with threads) |
| **Thread Safety** | Guaranteed (Using atomic updates) | Guaranteed (Using intrinsic locks) |
| **Deadlock** | **No** (Lock-free) | **Yes** (Possible if locks are misused) |
| **Usage** | Simple variable updates (e.g., counters) | Complex critical sections |

**💡 When to Use:**

* **CAS**: High-concurrency tasks (e.g., counters, queues).
* **Synchronized**: Complex operations requiring **multiple steps**.

**📌 Internal Working of CAS in Java**

1. Uses **CPU-level instructions** (e.g., cmpxchg in x86).
2. **Unsafe Class**: Java's sun.misc.Unsafe provides **low-level memory access**.
3. **Atomic Classes**: CAS powers AtomicInteger, AtomicBoolean, etc.

**✅ CAS in AtomicInteger Source Code**

public final boolean compareAndSet(int expected, int newValue) {

return unsafe.compareAndSwapInt(this, valueOffset, expected, newValue);

}

* **unsafe**: Directly interacts with the JVM.
* **compareAndSwapInt**: Atomically compares and sets the value.

**📌 Limitations of CAS**

1. **ABA Problem**: Value changes from A → B → A – CAS cannot detect intermediate changes.  
   ✅ **Solution**: Use **AtomicStampedReference** to track versions.
2. **Spin Waiting (CPU Usage)**: CAS **spins** (retries) when contention is high, wasting CPU cycles.  
   ✅ **Solution**: Use a **backoff** strategy or **adaptive spinning**.

**✅ ABA Problem Example**

import java.util.concurrent.atomic.AtomicStampedReference;

public class ABAProblemExample {

public static void main(String[] args) {

AtomicStampedReference<Integer> atomicRef = new AtomicStampedReference<>(100, 1);

int stamp = atomicRef.getStamp();

atomicRef.compareAndSet(100, 200, stamp, stamp + 1); // First update

System.out.println("Current Value: " + atomicRef.getReference() + ", Stamp: " + atomicRef.getStamp());

}

}

**📌 Popular Java Classes Using CAS**

| **Class** | **Purpose** |
| --- | --- |
| AtomicInteger | Atomic integer operations |
| AtomicBoolean | Atomic boolean operations |
| AtomicLong | Atomic long operations |
| AtomicReference | Atomic object reference |
| AtomicStampedReference | Atomic reference with versioning |
| ConcurrentLinkedQueue | Lock-free queue |

**📌 Common CAS-Based Interview Questions**

1. **What is Compare-And-Swap (CAS) and how does it work?**
2. **Explain how CAS prevents race conditions.**
3. **What is the ABA problem in CAS? How do you solve it?**
4. **Compare CAS with synchronized – which is better and when?**
5. **What is the role of Unsafe in CAS implementation?**
6. **When would you use AtomicStampedReference?**
7. **How does CAS improve performance in concurrent environments?**
8. **Explain the difference between compareAndSet() and weakCompareAndSet().**
   * **compareAndSet()** is **guaranteed** to succeed or fail.
   * **weakCompareAndSet()** may **fail spuriously** (useful in loops for faster execution).

**✅ Summary**

* **CAS** provides a **non-blocking** mechanism for updating variables in **multi-threaded** environments.
* **Efficient** for concurrent programs with **simple atomic updates**.
* **Drawbacks** include the **ABA problem** and **CPU overhead** during contention.
* Used extensively in **atomic classes** and **concurrent collections** in **Java**.

Would you like code examples for **ABA prevention** or **deep-dive into Unsafe**? 😊