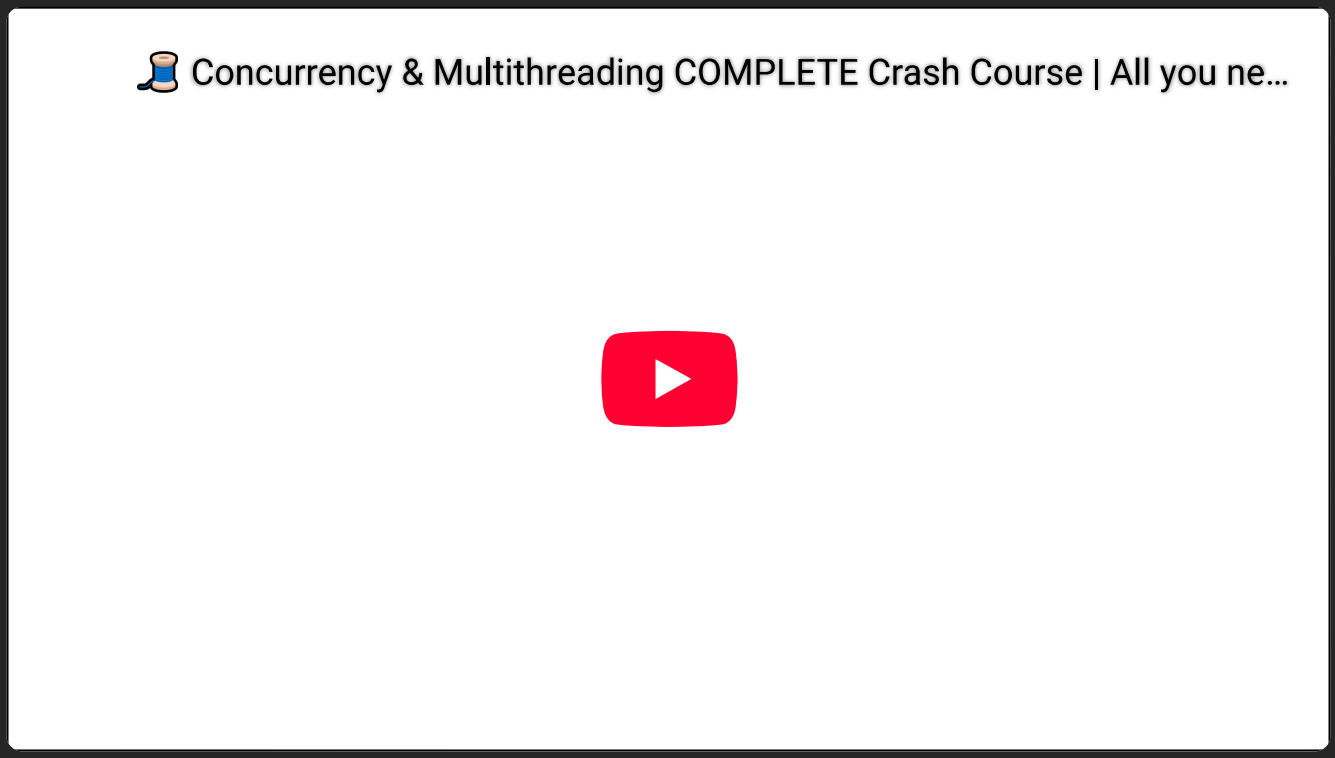
Search articles...

Sign In

Java Concurrent Collections



Topic Tags:

System Design LLD

Github Codes Link: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Java Concurrent Collections

Java Concurrent Collections are specialized thread-safe collection implementations designed for use in multithreaded environments. These collections provide better performance and scalability than their synchronized counterparts while ensuring thread safety.

Major Concurrent Collections:

1. ConcurrentHashMap
• HashMap :

By itself, a HashMap is not thread-safe. ❌ To use it in a concurrent context, you typically have to lock (synchronize) all operations externally. 🔒 This means that every operation (read or write) requires locking the entire map, which can be a performance bottleneck. 🐢

• ConcurrentHashMap 🧠⚙️:
This map is designed for concurrency. 💡 It splits the locking (or uses non-blocking techniques) so that multiple threads can operate on the map without needing to lock the entire data structure. 🔄🛡️ This leads to a much higher performance in multi-threaded environments, and you don't have to manage the locks manually. 🙌

Here's a breakdown of their differences, functionalities, and locking strategies 📝🔍:

How HashMap works (Internals) 🧬:

• HashMap internally uses an array of buckets to store entries. □ Each bucket can hold a linked list 🔗 (or a balanced tree 🌳 in later Java versions for better performance with many collisions) of Entry objects.

• When you put a key-value pair 🔑➡️📦, the hash code of the key is calculated 🔢, and this hash code is used to determine the bucket where the entry should be placed. 📍

• If multiple keys have the same hash code (hash collision) ⚠️, they are stored in the same bucket (as a linked list or tree). 🌲
• Operations like get, put, and remove involve calculating the hash code 🧮, finding the appropriate bucket □, and then traversing the linked list/tree within that bucket to find or modify the entry. 🔍✏️

How ConcurrentHashMap Works (Internals) 🛠️🔒:

How it works (Internals) 🔎:
ConcurrentHashMap employs a more sophisticated internal structure based on segments (in older Java versions up to Java 7) or buckets (in Java 8 and later). 🧱

Java 8 and later (Bucket-based) 🧰:

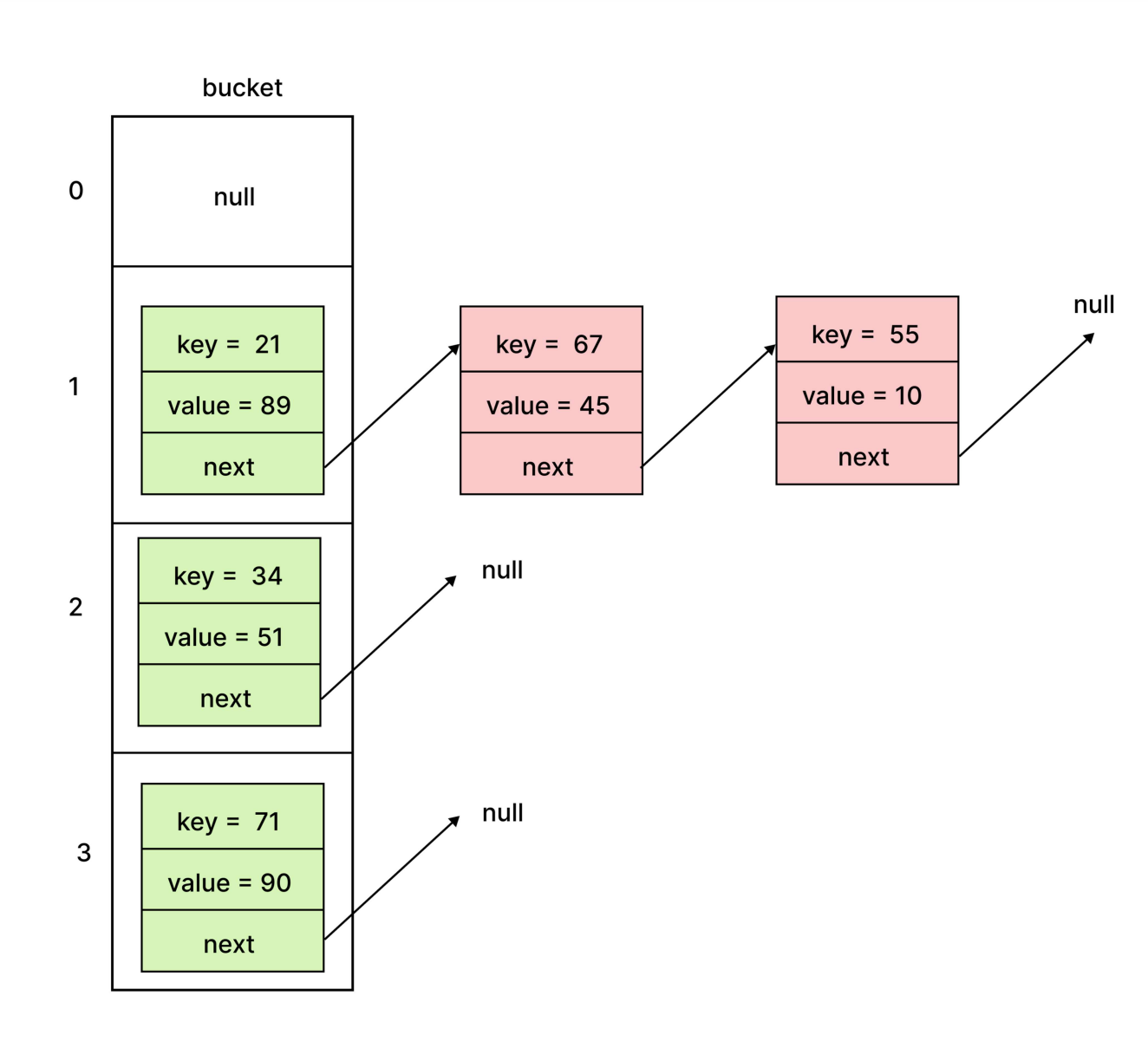
The internal structure is similar to HashMap (using an array of nodes/buckets). However, concurrency is achieved through finer-grained locking 🔒🔒 at the bucket level (using synchronized on the first node of a bucket during modifications) and optimistic locking techniques using Compare-and-Swap (CAS) operations for read operations and some structural modifications.
For read operations, most of the time, no locking is required. 👀⚡

ConcurrentHashMap vs. HashMap: How Locking Differs (Basic) 🛠️🔒::

The core difference in thread safety lies in how they manage access when multiple threads interact with the map.

HashMap achieveing Concurrency 🔒🧱:
• Locks the entire data structure. When any thread needs to modify the HashMap (e.g., add, remove, or even resize in some cases), it essentially needs exclusive access to the entire data structure.

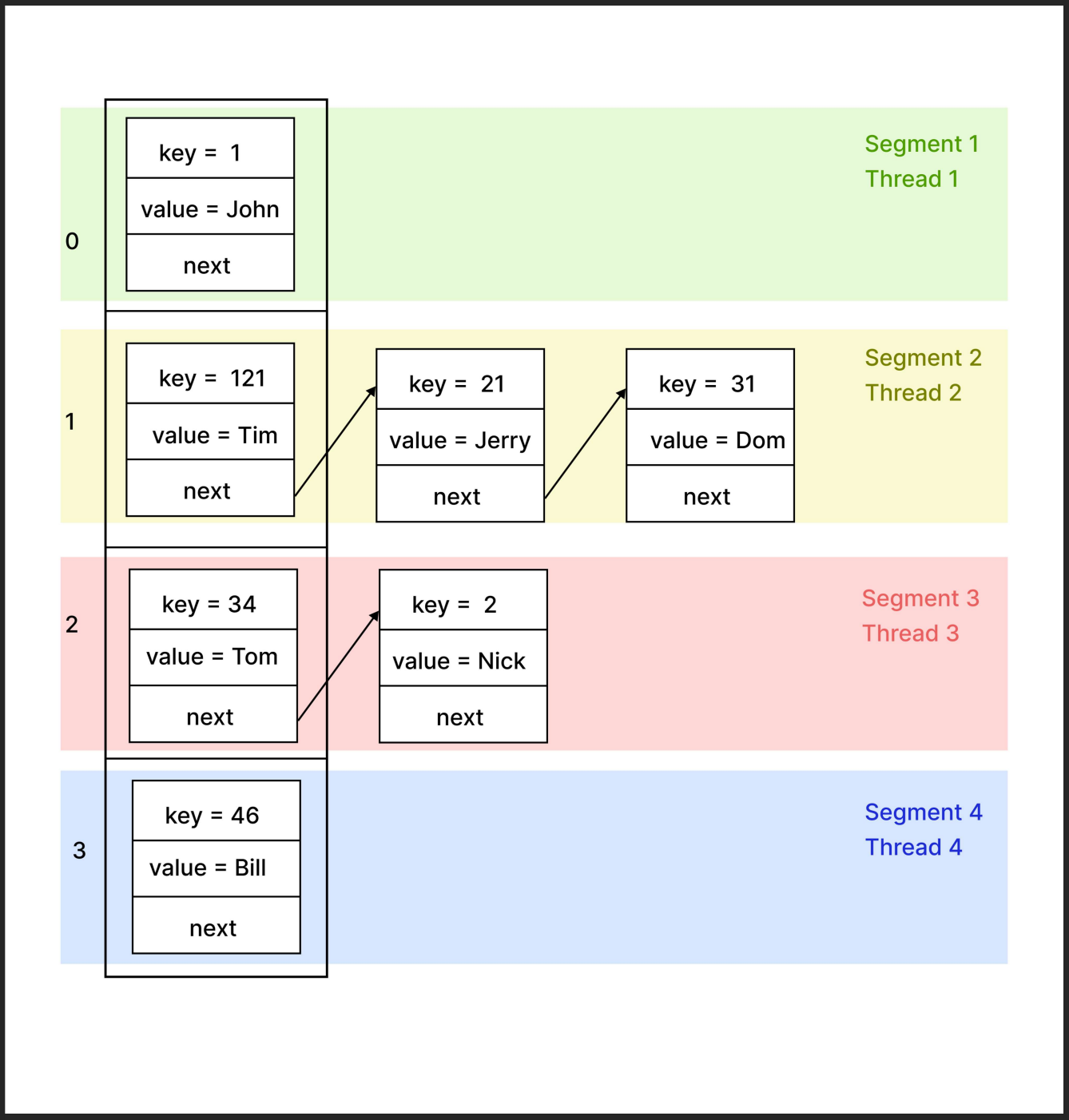
Imagine a single key to a treasure chest – only one person can hold the key and access anything inside at a time. This prevents multiple threads from making conflicting/non-conflicting changes simultaneously, and it also severely limits concurrency 🛑👥.



Example: If one thread is putting a new entry into a HashMap, all other threads that want to read from or write to the map have to wait until the first thread is finished.

ConcurrentHashMap:

• Locks in segments (or buckets). Instead of a single lock for everything, ConcurrentHashMap divides its internal data structure into multiple independent segments (in older versions) or individual buckets (in newer versions). Each segment or bucket has its own lock. Think of it as multiple smaller treasure chests, each with its own key. Different threads can hold the keys to different chests and access them concurrently without blocking each other.



Example: If ConcurrentHashMap is divided into 16 segments, up to 16 different threads can potentially be performing write operations simultaneously, each on a different segment. Threads reading data generally don't even need to acquire a lock in the latest versions. This allows for much higher concurrency and better performance in multi-threaded applications compared to HashMap.

A high-performance thread-safe alternative to HashMap and Hashtable. 🗺️

Example:

Java

```java
public class MapExample {
  public static void main(String[] args) throws InterruptedException {
    // Create a non-thread-safe HashMap wrapped as a synchronized map.
    // Even though Collections.synchronizedMap provides basic thread safet
    // operations, you still need to manually synchronize when iterating o
    final Map<Integer, String> hashMap = Collections.synchronizedMap(new H

    // Create a ConcurrentHashMap which is thread-safe and designed for co
    final ConcurrentHashMap<Integer, String> concurrentMap = new Concurren

    // ----- Example 1: Using HashMap with manual locking -----
    Thread hashMapUpdater = new Thread(() -> {
      for (int i = 1; i <= 5; i++) {
        // No explicit manual lock is needed for put() as synchronizedMap
        hashMap.put(i, "Value " + i);
        try {
          Thread.sleep(50); // simulate some processing time
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
        }
      }
    });

    Thread hashMapIterator = new Thread(() -> {
      // Wait a short while so that some entries are added.
      try {
        Thread.sleep(25);
      } catch (InterruptedException e) {
```

```
        Thread.currentThread().interrupt();
      }

      // When iterating over a synchronizedMap, you must lock the map manu
      synchronized (hashMap) {
        for (Map.Entry<Integer, String> entry : hashMap.entrySet()) {
          System.out.println(
              "hashMap Iteration - Key: " + entry.getKey() + ", Value: " +
        }
      }
    });

    // Start both threads and wait for them to finish.
    hashMapUpdater.start();
    hashMapIterator.start();
    hashMapUpdater.join();
    hashMapIterator.join();
    System.out.println("Final hashMap: " + hashMap);

    // ----- Example 2: Using ConcurrentHashMap -----
    Thread concurrentMapUpdater = new Thread(() -> {
      for (int i = 1; i <= 5; i++) {
        concurrentMap.put(i, "Value " + i);
        try {
          Thread.sleep(50); // simulate some processing time
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
        }
      }
    });

    Thread concurrentMapIterator = new Thread(() -> {
      // Wait a short while so that some entries are added.
      try {
        Thread.sleep(25);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }

      // With ConcurrentHashMap, iteration is safe without any external sy
      for (Map.Entry<Integer, String> entry : concurrentMap.entrySet()) {
```

```
            System.out.println(
                "concurrentMap Iteration - Key: " + entry.getKey() + ", Value:
        }
    });

    // Start both threads and wait for them to finish.
    concurrentMapUpdater.start();
    concurrentMapIterator.start();
    concurrentMapUpdater.join();
    concurrentMapIterator.join();
    System.out.println("Final concurrentMap: " + concurrentMap);
  }
}
```

Output :

```
hashMap Iteration - Key: 1, Value: Value 1
hashMap Iteration - Key: 2, Value: Value 2
Final hashMap: {1=Value 1, 2=Value 2, 3=Value 3, 4=Value 4, 5=Value 5}
concurrentMap Iteration - Key: 1, Value: Value 1
concurrentMap Iteration - Key: 2, Value: Value 2
concurrentMap Iteration - Key: 3, Value: Value 3
concurrentMap Iteration - Key: 4, Value: Value 4
concurrentMap Iteration - Key: 5, Value: Value 5
Final concurrentMap: {1=Value 1, 2=Value 2, 3=Value 3, 4=Value 4, 5=Value 5}
```

• Locking with HashMap 🔓🔓:
You need to explicitly lock the map (and often the iteration) to ensure thread safety. This is what the comment “trying to lock the HashMap Not really” refers to—simply wrapping it with synchronization works, but it is coarse and less efficient 👀⚡.

• ConcurrentHashMap 🔓🔓:
It handles concurrent access more efficiently without the need for external locking, thanks to its internal design
optimized for simultaneous operations 🧱.

2. CopyOnWriteArrayList:

CopyOnWriteArrayList is a thread-safe variant of ArrayList that creates a fresh copy of the underlying array for every modification operation. This unique approach ensures thread

safety while providing non-blocking read operations, making it particularly well-suited for specific concurrency scenarios. 🔒

Both ArrayList and CopyOnWriteArrayList in Java implement the List interface, providing ordered collections of elements. 📚 However, they differ significantly in their handling of concurrent access, particularly in their approach to thread safety, making them suitable for different scenarios. ⚖️

Here's a breakdown of their differences, functionalities, and thread-safety mechanisms: 🧩🔍

How ArrayList works (Internals): 📋

• ArrayList internally uses a dynamic array to store elements. ➕ When the array reaches its capacity, a new larger array is created, and all elements are copied to the new array. 📦

• When you add or remove elements, the array is modified directly, and in multi-threaded environments, this can lead to data corruption if multiple threads modify the array concurrently. ⚠️🧵

• Operations like get(), add(), and remove() directly access or modify the underlying array without any built-in synchronization. 🚫🔒\

• If one thread is iterating through an A
rrayList while another thread modifies it, a ConcurrentModificationException is likely to be thrown, as the iterator detects that the collection has been modified during iteration. 🚨🔁

How CopyOnWriteArrayList Works (Internals): 📝

• CopyOnWriteArrayList maintains an immutable array that is only changed by creating and reassigning a new array instance. 🔁📄

• When a modification operation (add, remove, set) is performed, the entire array is copied to a new array with the modification applied, and the reference to the array is atomically updated to point to the new array. ⚙️🔄

• Read operations (like get(), size(), contains()) operate on the current array reference without any locking, providing non-blocking reads. 🚀👓

• Iterators created from the list work on a snapshot of the array at the time the iterator was created, making them immune to concurrent modifications and eliminating the need to throw ConcurrentModificationException. 🛡️🧭

CopyOnWriteArrayList vs. ArrayList: How Thread Safety Differs (Basic): 🔄🔧

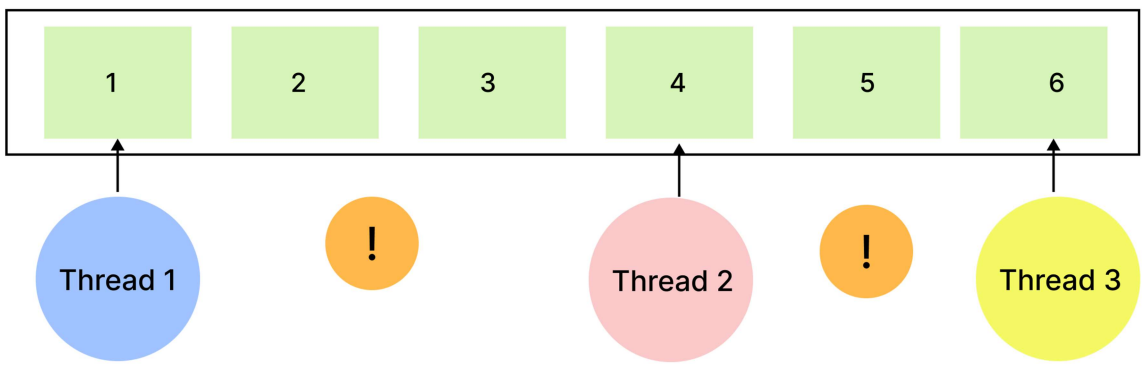
The core difference in thread safety lies in how they manage access when multiple threads interact with the list. 👥🧠

ArrayList 📜:

Not thread-safe by default. 🚫 When multiple threads access and modify an ArrayList concurrently, it can lead to data corruption or inconsistent states. 💥 Imagine multiple people trying to modify a document simultaneously without any coordination – the result would be chaos. 📝😵

Solutions for thread safety with ArrayList typically involve external synchronization, such as using Collections.synchronizedList() which essentially locks the entire list during any operation, severely limiting concurrency. 🔒👫

Example: If one thread is adding elements to an ArrayList while another thread is iterating over it, a ConcurrentModificationException will likely be thrown, or worse, the iteration might miss elements or see partially updated states. ⚠️📉



Problems: Direct concurrent access

Multiple threads directly modify the same underlying array without coordination, causing data consumption

X Can cause concurrentModificationException

X May lead to data corruption and race conditions

X Required external synchronization for thread safety

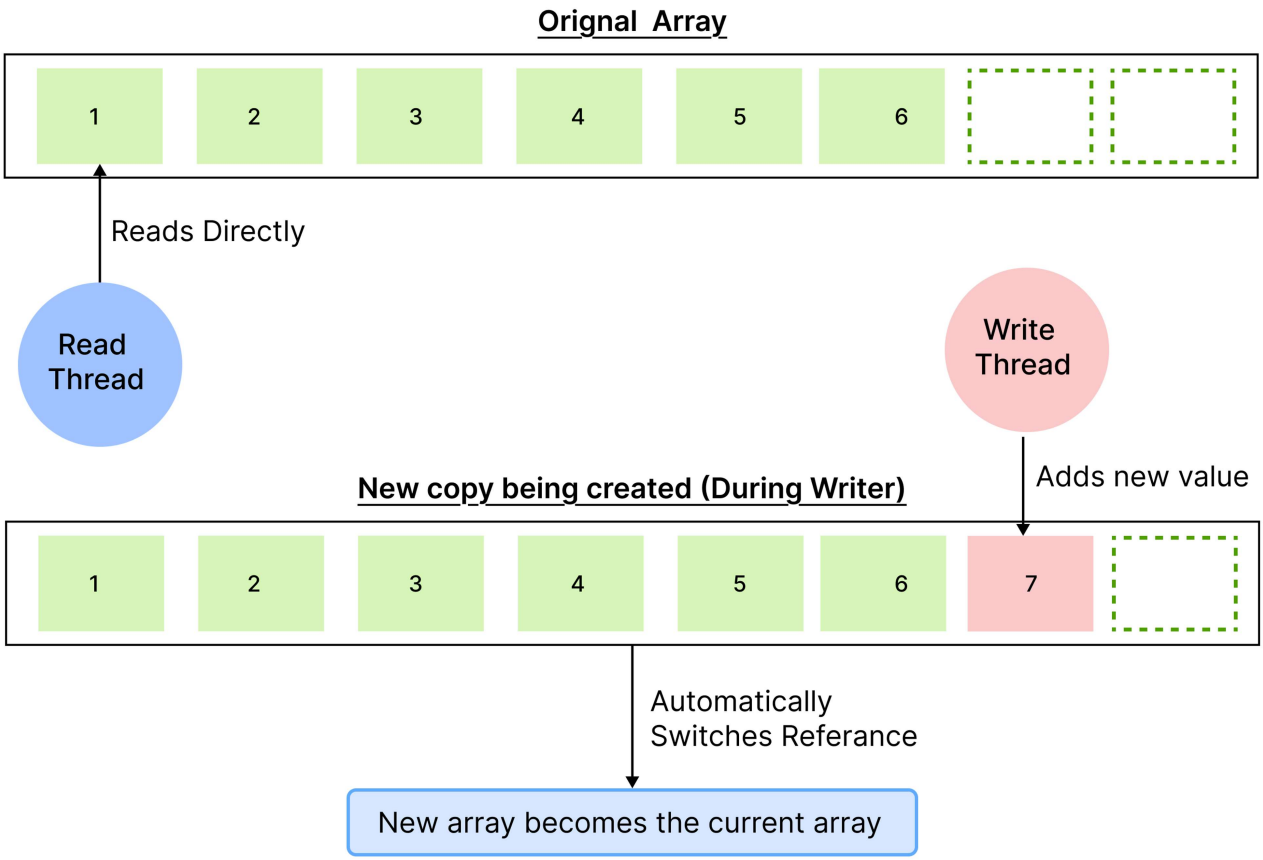
CopyOnWriteArrayList:

• Thread-safe by design, using a copy-on-write strategy. 🔐📝 Think of it as making a complete photocopy of a document 📄📠 whenever any changes are needed, then replacing the original with the updated copy once the changes are complete. ✅📄

• Modification operations are synchronized internally 🔁🔒, ensuring that only one thread can modify the list at a time. 🕵️ However, this synchronization is only for the brief moment needed to create and swap the array references. 🔄🧠

• Read operations require no synchronization ❌🔒 and can occur concurrently with other reads and even during write operations (though they will see the state before any concurrent writes). 👀⚡

Example 🧪:
If one thread is iterating over a CopyOnWriteArrayList while another thread adds elements ➕, the iteration will proceed unaffected over the original array snapshot 🖼️, and no exceptions will be thrown 🚫⚠️. The additions will be visible only to iterations started after the modification completed. 🕒🕶️



Benefits of Copy-on-write Stratergy

✅ Read Operation proceed without blocking

✅ No ConcurrentModificationException during iteraton

✅ Thread-safe without external synchronization

Java

```java
public class ListExample {
  public static void main(String[] args) throws InterruptedException {
    // --- Example 1: Using a plain ArrayList (Not Thread-Safe) ---
    // Initialize the ArrayList with some values.
    final List<Integer> arrayList = new ArrayList<>();
    arrayList.add(1);
    arrayList.add(2);
    arrayList.add(3);
```

```
9
10      Thread arrayListWriter = new Thread(() -> {
11        try {
12          // Sleep to let the reader start iterating
13          Thread.sleep(50);
14          for (int i = 4; i <= 6; i++) {
15            System.out.println("ArrayList Writer adding: " + i);
16            arrayList.add(i);
17            Thread.sleep(50);
18          }
19        } catch (InterruptedException e) {
20          Thread.currentThread().interrupt();
21        }
22      });
23
24      Thread arrayListReader = new Thread(() -> {
25        try {
26          // Wait a bit so that writer thread starts its job
27          Thread.sleep(25);
28
29          // This loop is likely to throw ConcurrentModificationException
30          // because the list is modified during iteration.
31          for (Integer item : arrayList) {
32            System.out.println("ArrayList Reader read: " + item);
33            Thread.sleep(50);
34          }
35        } catch (InterruptedException e) {
36          Thread.currentThread().interrupt();
37        } catch (Exception e) {
38          // Catching ConcurrentModificationException or any other exception
39          System.out.println("Exception in ArrayList Reader: " + e);
40        }
41      });
42      System.out.println("Using ArrayList:");
43      arrayListWriter.start();
44      arrayListReader.start();
45      arrayListWriter.join();
46      arrayListReader.join();
47      // Print final state of ArrayList
48      System.out.println("Final ArrayList: " + arrayList);
49
```

```
50      // --- Example 2: Using a CopyOnWriteArrayList (Thread-Safe) ---
51      // Initialize the CopyOnWriteArrayList with the same initial values.
52      final CopyOnWriteArrayList<Integer> cowList = new CopyOnWriteArrayList
53      cowList.add(1);
54      cowList.add(2);
55      cowList.add(3);
56
57      Thread cowListWriter = new Thread(() -> {
58        try {
59          // Sleep to let the reader start iterating on the initial snapshot
60          Thread.sleep(50);
61          for (int i = 4; i <= 6; i++) {
62            System.out.println("CopyOnWriteArrayList Writer adding: " + i);
63            cowList.add(i);
64            Thread.sleep(50);
65          }
66        } catch (InterruptedException e) {
67          Thread.currentThread().interrupt();
68        }
69      });
70
71      Thread cowListReader = new Thread(() -> {
72        try {
73          // Wait a little so that the writer thread may add new elements
74          // (but the iterator is created before those new elements are adde
75          Thread.sleep(25);
76          // Iterating here is safe; the iterator will see a snapshot of the
77          for (Integer item : cowList) {
78            System.out.println("CopyOnWriteArrayList Reader read: " + item);
79            Thread.sleep(50);
80          }
81        } catch (InterruptedException e) {
82          Thread.currentThread().interrupt();
83        }
84      });
85      System.out.println("nUsing CopyOnWriteArrayList:");
86      cowListWriter.start();
87      cowListReader.start();
88      cowListWriter.join();
89      cowListReader.join();
90      // Print final state of CopyOnWriteArrayList
```

```
        System.out.println("Final CopyOnWriteArrayList: " + cowList);
    }
}
```

Output :

```
Using ArrayList:
ArrayList Reader read: 1
Exception in ArrayList Reader: java.util.ConcurrentModificationException
ArrayList Writer adding: 4
ArrayList Writer adding: 5
ArrayList Writer adding: 6
Final ArrayList: [1, 2, 3, 4, 5, 6]

Using CopyOnWriteArrayList:
CopyOnWriteArrayList Reader read: 1
CopyOnWriteArrayList Reader read: 2
CopyOnWriteArrayList Reader read: 3
CopyOnWriteArrayList Writer adding: 4
CopyOnWriteArrayList Writer adding: 5
CopyOnWriteArrayList Writer adding: 6
Final CopyOnWriteArrayList: [1, 2, 3, 4, 5, 6]
```

In this example, we first create an ArrayList (which is not thread-safe) and a CopyOnWriteArrayList (designed for concurrent operations). Then we spawn a writer thread that adds elements after a brief pause and a reader thread that iterates over the list. We add a slight sleep in both threads to simulate parallel behavior.

Note that with the ArrayList, you'll likely get a ConcurrentModificationException because its iterator is not safe when the list is modified concurrently. The CopyOnWriteArrayList, on the other hand, creates a snapshot at the time of the iterator's creation so that the reader sees only the elements present at that moment—even if the list is updated later.

3. ConcurrentLinkedQueue:

ConcurrentLinkedQueue is a thread-safe implementation of a queue (a FIFO data structure) that uses a linked-node structure and non-blocking algorithms to achieve high concurrency. 🔄🧵⚙️ It allows multiple threads to safely add and remove elements without excessive locking, making it ideal for high-throughput, multi-producer/multi-consumer scenarios. 🚀👥👥

Both LinkedList (when used as a queue) and ConcurrentLinkedQueue in Java implement the Queue interface, providing first-in-first-out (FIFO) operations. 📑➡️ However, they differ significantly in their handling of concurrent access, particularly in their implementation of thread safety, making them suitable for different scenarios. ⚖️🔐

Here's a breakdown of their differences, functionalities, and thread-safety mechanisms:

How LinkedList (as a Queue) works (Internals): 📋🔗

• LinkedList internally uses a doubly-linked list structure, with each node containing references to both the previous and next nodes in the sequence. ↔️📦

• When elements are added (offered) to the queue ➕📥, they are appended to the tail of the list. When elements are removed (polled) from the queue ➖📤, they are taken from the head of the list.
• Operations like offer(), poll(), and peek() directly modify or access the underlying linked structure without any built-in synchronization. 🔧🚫🔒

• In multi-threaded environments, concurrent modifications to a LinkedList can lead to data corruption, inconsistent states, or ConcurrentModificationException during iteration. ⚠️🧨

How ConcurrentLinkedQueue Works (Internals): 🧠🔗

• ConcurrentLinkedQueue uses a singly-linked list structure, with each node containing a reference to the next node in the sequence. ➡️📦

• The implementation is based on non-blocking algorithms using atomic compare-and-swap (CAS) operations 🎚️🔁, which allow threads to make progress without explicit locks. 🔓🤜

• When a thread wants to add an element (offer) ➕, it attempts to set the next reference of the current tail node to the new node. If another thread has modified the queue in the meantime, the operation is retried. 🔁📥

• Similarly, when removing an element (poll) ➖, the operation attempts to set the head reference to the next node. Again, if concurrent modifications have occurred, the operation is retried. 🔁📤

This "optimistic" approach allows high throughput in multi-threaded environments, as threads don't have to wait for locks to be released. 🤜🔄🧵

What is Compare and Swap (CAS)? 🤔🔁

Compare and swap is a hardware-supported atomic operation that works as follows: 🖥️⚙️

• Compare: The operation reads a value from a memory location and compares it with an expected value. 📄🔍

• Swap: If and only if the current value in memory equals the expected value, it writes a new value into that memory location. 🔄✅

In Java, the method compareAndSet from classes like AtomicReference encapsulates this operation. 📦💉 The method returns a boolean indicating whether the swap was successful. ✔️❌

Java

```
AtomicReference<Node> ref = new AtomicReference<>(currentNode);
boolean isUpdated = ref.compareAndSet(currentNode, newNode);
```

In this code: 💻🔤
• If ref still holds the currentNode, it gets updated to newNode, and compareAndSet returns true. 🔁✅
• If ref has been updated by another thread in the meantime (i.e., the value is different from currentNode), the method returns false, indicating that the operation should be retried. 🔁🚫

ConcurrentLinkedQueue vs. LinkedList: How Thread Safety Differs (Basic): 🔒

The core difference in thread safety lies in how they manage access when multiple threads interact with the queue. 🧵🔄

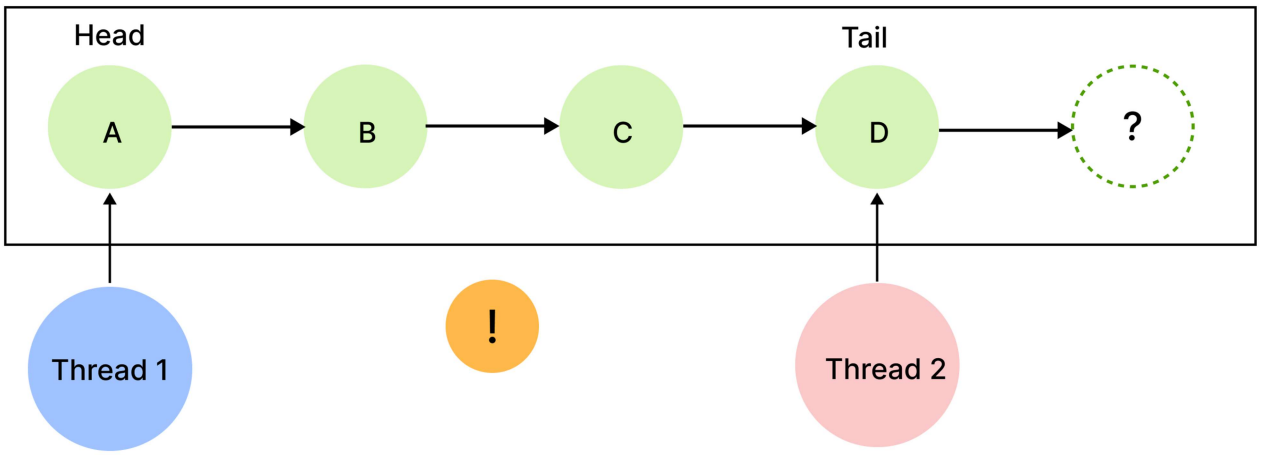
LinkedList: 🧱

• Not thread-safe by default. 🚫🔐 When multiple threads access and modify a LinkedList concurrently, it can lead to data corruption or inconsistent states. ⚠️💥 Imagine several people trying to form and manage a single line without any coordination – some might end up in the wrong position, or even be forgotten entirely. 🧍‍♀️❓

• Solutions for thread safety with LinkedList typically involve external synchronization, such as using Collections.synchronizedList() 🔄🛡️ which essentially locks the entire list during any operation, severely limiting concurrency. 🔐🐢

Example: 📌
If one thread is adding elements to a LinkedList ➕ while another thread is removing elements ➖, the list's internal pointers might become corrupted, leading to data loss 🗑️ or

even infinite loops during traversal. 🔁♾️



Potential Concurrency Problems

When thread 1 removes from head while Thread 2 adds to talk the shared data structure can become corrupted

X No built-in sybchronization

X Can cause poniter corruption data loss, or exception

ConcurrentLinkedQueue:

How ConcurrentLinkedQueue Achieves Concurrency 🚀🧠

1. Lock-Free Design and Optimistic Concurrency: 🔓✨

ConcurrentLinkedQueue is designed as a lock-free data structure. Instead of using traditional locking mechanisms that block threads, it operates optimistically. 🤞🛠️
This means that threads assume that collisions (two threads modifying the structure at the same time) are rare. 💥

When a thread performs an update—such as adding ➕ or removing ➖ an element—it uses CAS to atomically check whether the part of the data structure it is trying to modify is still in the expected state. ✔️📍

If another thread has made a change in the meantime, the CAS fails, and the thread can retry the operation. 🔁❌✅

2. Structural Overview: 🏗️🔧
• Linked Nodes: 📦
The queue is made up of nodes that are connected by pointers (typically, a next pointer). 🔗

• Head and Tail Pointers: 🎯🔚
There are references to the head and tail of the queue that are updated when elements are removed or added. 🔄

• CAS on Pointer Updates: 🧪🔁
When a thread wants to add an element (enqueue), it does something conceptually similar to: 🧵➕📌

Java

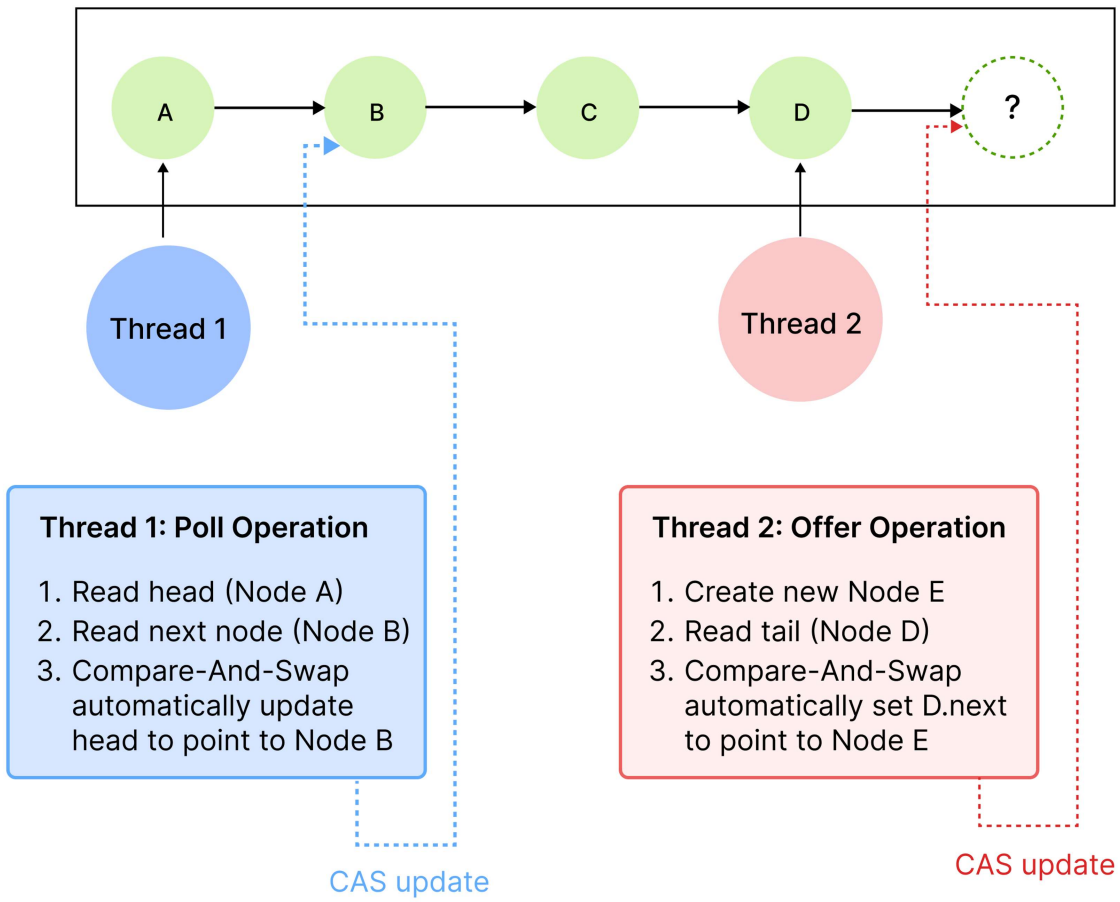
```java
// Pseudocode for enqueue operation in a concurrent queue
Node last = tail;
Node newNode = new Node(element);
// Try to link the new node after the last node atomically.
// If last.next is still null (meaning no one has updated it),
// then atomically change it to point to newNode.
if (CAS(last.next, null, newNode)) {
    // Successfully appended newNode; now try to update the tail pointer too
    CAS(tail, last, newNode);
} else {
    // If CAS fails, another thread interfered.
    // Retry the operation or help update the tail pointer.
}
```

• Retry on Failure: 🔁❌✅
If the CAS fails because another thread made a concurrent update, the current thread simply retries the operation with the updated value. 🧠📌

3. Efficiency without Copying: ⚡📉
• Unlike a CopyOnWriteArrayList—which creates a new copy 📄📄 of the underlying array on every write—the ConcurrentLinkedQueue only modifies pointers (or references). 🔗
• This makes updates generally more efficient, especially when writes are frequent. ⚙️👉
• There is no overhead of copying the entire data structure, and memory usage remains more stable. 🧠💾

Example: 🧪

If multiple threads are simultaneously adding elements to ➕ and removing elements from ➖ a ConcurrentLinkedQueue, the queue's internal structure remains consistent, and all operations eventually complete successfully without blocking each other. 🔓❌✅



Benefits of Concurrent List Queue

- ✅ Both Operations proceed without blocking
- ✅ Non Blocking algorithm
- ✅ No locks required
- ✅ High throughput
- ✅ Multiple Producers/consumers

Java

```java
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.ConcurrentLinkedQueue;
public class QueueComparison {
    public static void main(String[] args) throws InterruptedException {
        // Demonstrate issues with LinkedList in a multi-threaded environm
        System.out.println("--- LinkedList Example (Not Thread-Safe) ---")

        Queue<Integer> linkedListQueue = new LinkedList<>();

        // Create producer and consumer threads
        Thread producerLinkedList = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                linkedListQueue.offer(i);
            }
            System.out.println("Producer finished adding 1000 items to Lin
        });

        Thread consumerLinkedList = new Thread(() -> {
            int count = 0;
            while (count < 1000) {
                Integer item = linkedListQueue.poll();
                if (item != null) {
                    count++;
                }
            }
            System.out.println("Consumer processed " + count + " items fro
        });

        // Start both threads
        producerLinkedList.start();
        consumerLinkedList.start();

        // Wait for completion
        producerLinkedList.join();
        consumerLinkedList.join();

        System.out.println("LinkedList size after operations (should be 0)
        System.out.println("Note: LinkedList might have unexpected behavio
        System.out.println();
```

```
41
42          // Demonstrate ConcurrentLinkedQueue in a multi-threaded environme
43          System.out.println("--- ConcurrentLinkedQueue Example (Thread-Safe
44
45          ConcurrentLinkedQueue<Integer> concurrentQueue = new ConcurrentLin
46
47          // Multiple producer threads
48          Thread producer1 = new Thread(() -> {
49              for (int i = 0; i < 500; i++) {
50                  concurrentQueue.offer(i);
51              }
52              System.out.println("Producer 1 finished adding 500 items ✅");
53          });
54
55          Thread producer2 = new Thread(() -> {
56              for (int i = 500; i < 1000; i++) {
57                  concurrentQueue.offer(i);
58              }
59              System.out.println("Producer 2 finished adding 500 items ✅");
60          });
61
62          // Multiple consumer threads
63          Thread consumer1 = new Thread(() -> {
64              int count = 0;
65              while (count < 500) {
66                  Integer item = concurrentQueue.poll();
67                  if (item != null) {
68                      count++;
69                  }
70              }
71              System.out.println("Consumer 1 processed 500 items ✅");
72          });
73
74          Thread consumer2 = new Thread(() -> {
75              int count = 0;
76              while (count < 500) {
77                  Integer item = concurrentQueue.poll();
78                  if (item != null) {
79                      count++;
80                  }
81              }
```

```
            System.out.println("Consumer 2 processed 500 items ✅");
        });

        // Start all threads
        producer1.start();
        producer2.start();
        consumer1.start();
        consumer2.start();

        // Wait for completion
        producer1.join();
        producer2.join();
        consumer1.join();
        consumer2.join();

        System.out.println("ConcurrentLinkedQueue size after operations (s
    }
}
```

Output :

```
--- LinkedList Example (Not Thread-Safe) ---
Producer finished adding 1000 items to LinkedList
Consumer processed 1000 items from LinkedList
LinkedList size after operations (should be 0): 12
Note: LinkedList might have unexpected behavior or exceptions in concurrent scenari

--- ConcurrentLinkedQueue Example (Thread-Safe) ---
Producer 1 finished adding 500 items ✅
Producer 2 finished adding 500 items ✅
Consumer 1 processed 500 items ✅
Consumer 2 processed 500 items ✅
ConcurrentLinkedQueue size after operations (should be 0): 0
```

In an ideal, synchronized world you'd expect the consumer to remove every one of the 1000 items so that the final size is 0. However, because LinkedList is not thread-safe, concurrent calls to offer() and poll() can lead to race conditions that corrupt the internal state. This may result in some items never being removed or even being "lost" during insertion. In an unexpected run, you might see a final size greater than 0 (for example, 100, 250, etc.) because the unsynchronized modifications can overwrite or miss updates due to conflicting interleavings.

Because you're not using an iterator, you won't see a ConcurrentModificationException in this code. The exception typically occurs when you iterate over a collection (using an iterator or for-each loop) while it's being modified. In our example, you use offer() and poll(), which directly manipulate the LinkedList without throwing that specific exception even though the operations are unsynchronized and unsafe.

4. Blocking Queue Implementations: 🧱📦
Blocking Queues are thread-safe queue implementations that support operations which wait (block) ⏳ for the queue to become non-empty when retrieving elements, or wait for space to become available when adding elements. They are a fundamental building block for producer-consumer patterns in concurrent applications. ⏱️👷👷

How Blocking Queues Work (Internals): 🧠

Blocking queues provide a thread-safe mechanism for exchanging data between threads, typically in producer-consumer scenarios. 🧵🔁
Unlike non-blocking concurrent collections like ConcurrentLinkedQueue, blocking queues explicitly coordinate between threads using one of two fundamental approaches:

1. 🔐 Intrinsic Locking:
Most blocking queue implementations use locks (such as ReentrantLock) internally to synchronize access to the underlying data structure. 🧱🔒

2. ⏳ Condition Variables:
Conditions are used in conjunction with locks to allow threads to wait efficiently when they cannot proceed (e.g., when the queue is full or empty) and to be notified when they can continue. 📣⚙️

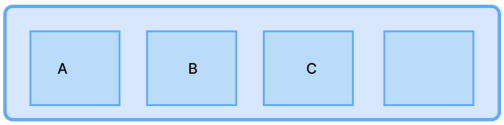
🧑🍳 When a producer thread tries to add an item to a full queue using put(), the thread is suspended until space becomes available. 🛑📥

🧑🔧 Similarly, when a consumer thread tries to take an item from an empty queue using take(), the thread waits until an item is added. 🛑📤

Common Blocking Queue Implementations: 📋

Blocking Queue Implementations

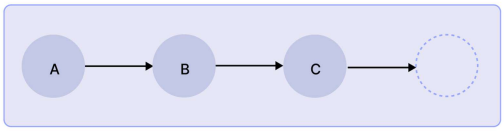
ArrayBlockingQueue



fixed capacity array (bounded)

- Fixed capacity specified at creation
- FIFO ordering
- Optional fairness policy

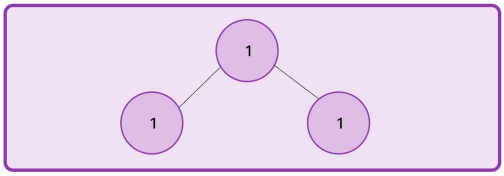
LinkedBlockingQueue



Linked nodes (optionally bounded)

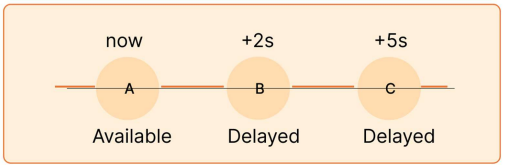
- Optionally bounded capacity
- FIFO ordering
- Hgher throughput than array-based

ArrayBlockingQueue



- Unbounded capacity
- priority ordering
- uses natural order or comparator

DelayQueue



- Unbounded queue of delayed elements
- Elements available only after delay expires
- Useful for time-based schedulling

Benefits Blocking Queue Implementations

✅ All implementations provide blocking operations

1. ArrayBlockingQueue: 🪣
• A bounded blocking queue backed by an array 🧱
• Fixed capacity specified at construction time 🔢
• Maintains FIFO (First-In-First-Out) ordering 🔁
• Optionally fair, ensuring threads are serviced in the order they requested access ⚖️

2. LinkedBlockingQueue: 🔗
• An optionally bounded blocking queue based on linked nodes 🧶
• Can be bounded or unbounded (with a capacity of Integer.MAX_VALUE) 🔓🔢
• Generally higher throughput than array-based queues but less predictable performance ⚡🔁

(A LinkedBlockingQueue uses separate locks for insertion (put) and removal (take), allowing more concurrent access and reducing contention compared to the single lock used in an ArrayBlockingQueue.) 🔓🔄
• Maintains FIFO ordering 🔁

3. PriorityBlockingQueue: 🏆
• An unbounded blocking queue that orders elements according to their natural ordering or a provided Comparator 🎚️📊
• Does not permit null elements 🚫
• Elements are dequeued in priority order, not FIFO 🎯
• Size is limited only by available memory 🧠💾

4. DelayQueue: 🕐
• An unbounded blocking queue of delayed elements 💤
• An element can only be taken from the queue when its delay has expired 📅
• The head of the queue is the element whose delay expired furthest in the past ⏳
• Useful for implementing time-based scheduling 📅📋

5. SynchronousQueue: 🤝
• A blocking queue with no internal capacity 🚫📦
• Each insert operation must wait for a corresponding remove operation by another thread, and vice versa 🔄
• Cannot peek at elements as there is no storage 👀❌
• Useful for direct hand-offs between threads 🔃

➡️ Non-blocking behavior means that operations on the queue (such as poll, peek, or add) return immediately without waiting—even if the queue is empty or being modified—using atomic, lock-free algorithms. ⚙️🚀

For instance, calling poll() on a ConcurrentLinkedQueue returns null instantly if the queue is empty instead of blocking the thread. 🧵👊

Java

```java
public class NonBlockingExample {
    public static void main(String[] args) {
        ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>()
        // Non-blocking poll: returns immediately with null because the queu
        Integer element = queue.poll();
        System.out.println("Polled element: " + element); // Expected output

        // Add an element; this operation is also non-blocking.
```

```
        queue.add(42);
        System.out.println("After adding: " + queue.poll()); // Expected out
    }
}
```

Both the poll() and add() methods execute without causing the thread to wait, illustrating the non-blocking nature of a ConcurrentLinkedQueue.

=> BlockingQueue (LinkedBlockingQueue) that demonstrates blocking behavior. Here, the main thread calls take() on an empty queue and waits (blocks) until another thread puts an element into it.

Java

```java
public class BlockingQueueDemo {
    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> queue = new LinkedBlockingQueue<>();
        // A new thread will add an element after a 2-second delay.
        new Thread(() -> {
            try {
                Thread.sleep(2000);
                queue.put(99); // put() will add element, unblocking take(
                System.out.println("Producer: Added 99");
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }).start();

        System.out.println("Consumer: Waiting to take an element...");

        // take() blocks until an element is available.
        Integer element = queue.take();
        System.out.println("Consumer: Took element " + element);
    }
}
```

Output :

```
Consumer: Waiting to take an element...
Producer: Added 99
Consumer: Took element 99
```

The call to queue.take()blocks until the producer thread adds the element 99, clearly demonstrating the blocking behavior of a BlockingQueue.

Key Features of Blocking Queues: 🛡️

• Thread Safety: All implementations are fully thread-safe, designed for concurrent access 🔒
• Blocking Operations: Support operations that block until they can proceed ⏱️
• Bounded Control: Can control memory usage with bounded implementations 📊
• Fairness Options: Some implementations support optional fairness policies ⚖️
• Timed Operations: Support for operations with timeout to avoid indefinite blocking ⏰
• Null Rejection: Do not permit null elements (which are used as sentinel values) ❌

Key Points to Remember:

✅ Choose the right collection based on your access patterns (read-heavy vs. write-heavy).
✅ ConcurrentHashMap is almost always preferred over Hashtable due to better performance.
✅ CopyOnWriteArrayList is ideal for rarely-modified, frequently-iterated lists.
✅ BlockingQueues are essential for producer-consumer patterns.
✅ Concurrent collections don't throw ConcurrentModificationException during iteration.

What is the main difference between fail-fast and fail-safe iterators?

• Fail-fast iterators (like in ArrayList, HashMap) throw ConcurrentModificationException if the collection is modified while iterating

• Fail-safe iterators (like in ConcurrentHashMap, CopyOnWriteArrayList) use a snapshot of the collection at the time the iterator was created, so they don't throw exceptions if the collection is modified during iteration

• Concurrent collections use fail-safe iterators

• Fail-fast is for detecting bugs, fail-safe is for thread safety

Conclusion 🎯

Java Concurrent Collections provide specialized, high-performance data structures for multithreaded environments. By choosing the right collection for your specific concurrency

needs, you can achieve better performance, scalability, and thread safety. 🚀🧩

Understanding the characteristics and use cases of different concurrent collections is essential for designing efficient concurrent applications. Whether you need maps, lists, queues, or sets, the Java concurrency framework offers optimized implementations for various concurrent access patterns. 🔄📊