1. Using `Collections.synchronizedList()`

java

Copy code

```java
import java.util.*;

public class SyncListExample {
    public static void main(String[] args) {
        List<String> list = Collections.synchronizedList(new ArrayList<>());

        // Multiple threads adding elements
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                list.add("T1-" + i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                list.add("T2-" + i);
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Iteration must be synchronized manually
        synchronized (list) {
            for (String s : list) {
                System.out.println(s);
            }
        }
    }
}
```

👉 Key point: **iteration requires explicit** `synchronized(list)`, otherwise you risk `ConcurrentModificationException`.

Note: Lock for both read and write

2. Using `CopyOnWriteArrayList`

```java
import java.util.concurrent.CopyOnWriteArrayList;

public class COWListExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();

        // Multiple threads adding elements
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                list.add("T1-" + i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                list.add("T2-" + i);
            }
        });

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Safe iteration without extra synchronization
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

👉 Key point: **iteration is safe without synchronization**, because it works on a snapshot copy (no `ConcurrentModificationException`).

Note: No lock read, lock for only write

✅ One-line summary:

- ** `synchronizedList(new ArrayList<>())` → Lock on **reads + writes**.
- ** `CopyOnWriteArrayList` → No lock on **reads**, lock + copy on **writes**.

- **Hash Table:** lock all the map
- **ConcurrentHashMap:** lock for a bucket of the map.

🔷 ConcurrentHashMap

- **Structure:** Hash table, but internally divided into **buckets/segments** (Java 7) → since Java 8, uses **node-level locking + CAS (compare-and-swap)** for higher concurrency.
- **Ordering: Unordered** (like `HashMap`).
- **Nulls:** ❌ **Does not allow null keys or null values** (to avoid ambiguity in concurrent lookups).
- **Thread Safety:** ✅ **Thread-safe**, much more scalable than `Hashtable` because:
  - `Hashtable` locks the entire map (`synchronized` on every method).
  - `ConcurrentHashMap` locks **only parts of the map** or uses CAS, allowing multiple threads to read/write at the same time.
- **Performance:**
  - Reads are **lock-free** (almost like a plain `HashMap`).
  - Writes are **concurrent**, with fine-grained locking.
  - Iterators are **weakly consistent** (they reflect the state of the map at some point, but don't throw `ConcurrentModificationException`).

👉 Best for **multi-threaded apps** where you need a high-performance, thread-safe map.