🔸 1. Basic Definition

Feature	ArrayList	LinkedList	Vector
Type	Resizable array	Doubly linked list	Resizable array
Package	`java.util`	`java.util`	`java.util`
Inheritance	Implements `List`	Implements `List`, `Deque`, `Queue`	Implements `List`

🔸 2. Internal Data Structure

Feature	ArrayList	LinkedList	Vector
Storage Mechanism	Dynamic array	Doubly linked list	Dynamic array
Memory Efficiency	Efficient for data	Efficient for structure	Less efficient (legacy)

🔸 3. Performance (Big O Complexity)

Operation	ArrayList	LinkedList	Vector
Add at End	O(1) (amortized)	O(1)	O(1) (synchronized)
Add at Middle/Start	O(n)	O(1) at start, O(n/2)	O(n)
Get (by index)	O(1)	O(n)	O(1)
Remove (by index)	O(n)	O(n)	O(n)
Remove First/Last	O(n)	O(1)	O(n)

🔸 4. Thread Safety

Feature	ArrayList	LinkedList	Vector
Synchronized	❌ No	❌ No	✅ Yes
Suitable for Multithreading	❌ Use `Collections.synchronizedList()` or `CopyOnWriteArrayList`	❌ Same	✅ Yes (but slower due to synchronization)

Feature	Vector	CopyOnWriteArrayList
Package	`java.util`	`java.util.concurrent`
Thread-Safety	✅ Yes (synchronized methods)	✅ Yes (by copy-on-write mechanism)
Synchronization Type	Synchronized on every operation	Copy on write (lock-free reads)
Performance (Reads)	❌ Slower (due to locking)	✅ Fast (no lock for reading)
Performance (Writes)	✅ Faster than CopyOnWriteArrayList	❌ Slower (copies entire array on each write)
Iterators	❌ Fail-fast (throws `ConcurrentModificationException`)	✅ Safe (no `ConcurrentModificationException`)
Use Case	Legacy synchronized list	Read-heavy, thread-safe list

✅ Internal Working

Feature	Vector	CopyOnWriteArrayList
Read operation	Uses synchronized blocks	Lock-free (reads from a stable array)
Write operation	Synchronized + modifies internal array	Creates a new copy of the array on each write
Iterators	Fail-fast	Safe snapshot iterators (returns a copy)

✅ Example Code

🔹 Vector Example

java
CopyEdit

```
import java.util.Vector;

public class VectorExample {
    public static void main(String[] args) {
        Vector<String> vector = new Vector<>();
        vector.add("Java");
        vector.add("Python");
        vector.add("C++");

        System.out.println(vector); // Output: [Java, Python, C++]
    }
}
```

🔹 CopyOnWriteArrayList Example

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

public class COWArrayListExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();
        list.add("Java");
        list.add("Python");
        list.add("C++");

        for (String lang : list) {
            list.add("Rust"); // Safe: No ConcurrentModificationException
            System.out.println(lang);
        }

        System.out.println(list); // Output includes Rust added during
iteration
    }
}
```

✅ Performance Comparison

Operation	Vector (synchronized)	CopyOnWriteArrayList (copy-on-write)
Read-heavy tasks	❌ Slower	✅ Much faster
Write-heavy tasks	✅ Better	❌ Expensive (full array copy)
Iteration safety	❌ Fail-fast	✅ No risk (uses snapshot)

✅ When to Use

- **Use `Vector`** if:
    - You are working with legacy code.
    - You want synchronized access and are okay with locks.
- **Use `CopyOnWriteArrayList`** if:
    - You have many more reads than writes.
    - You want **lock-free reads** with thread-safe iteration.
    - You want to **avoid** `ConcurrentModificationException`.

✅ Summary Table

Feature	Vector	CopyOnWriteArrayList
Thread-safe	✅ Yes	✅ Yes
Locking	Synchronized methods	No locking for read operations
Read performance	❌ Slower	✅ Faster
Write performance	✅ Faster	❌ Slower (copies array)
Iterator safety	❌ Fail-fast	✅ Safe snapshot
Best for	Balanced read/write	Read-heavy use cases
ConcurrentModificationException	✅ Can occur	❌ Never occurs

🔁 `ArrayList` vs `Vector` — Key Differences (in Multithreading)

Feature	`ArrayList`	`Vector`
Thread Safety	❌ Not synchronized (not thread-safe)	✅ Synchronized (thread-safe)
Performance	🚀 Faster (no overhead of locks)	🐢 Slower (synchronization overhead)
Usage in Threads	Use only with **external sync**	Safe for **multiple threads**

Grow Behavior	Grows by 50%	Grows by 100% (doubles in size)

✅ Vector is Synchronized

Every method in `Vector` is synchronized internally:

```
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
```

So when **multiple threads** access a `Vector`, only **one thread can modify it at a time**, which prevents data corruption but reduces performance due to locking.

❌ ArrayList is NOT Synchronized

If multiple threads access an `ArrayList` and at least one of them modifies it, **external synchronization** is required to prevent `ConcurrentModificationException` or inconsistent state:

```
List<Integer> list = Collections.synchronizedList(new ArrayList<>());
```

Or manual lock:

```
synchronized (arrayList) {
    arrayList.add(10);
}
```

🔁 Example of Thread Issue with ArrayList

```
List<Integer> list = new ArrayList<>();

Runnable task = () -> {
    for (int i = 0; i < 1000; i++) {
        list.add(i); // ❌ not thread-safe
    }
};

Thread t1 = new Thread(task);
Thread t2 = new Thread(task);
t1.start();
t2.start();
```

Above code may result in:

- Missing values
- `ConcurrentModificationException`
- Corrupt memory due to race conditions

✅ Safer with Vector

```
List<Integer> list = new Vector<>();

// Same task as above, but Vector is synchronized internally
```

This will work without errors, but might be slower due to locking.

💡 Better Alternative: `CopyOnWriteArrayList` or `Collections.synchronizedList`

Instead of using `Vector`, modern Java developers prefer:

```
List<Integer> list = new CopyOnWriteArrayList<>(); // Thread-safe, high read performance

List<Integer> list = Collections.synchronizedList(new ArrayList<>()); // Synchronized wrapper
```

🔚 Summary

Use Case	Recommended Collection
Single-threaded (no concurrency)	`ArrayList`
Multi-threaded (safe reads/writes)	`Collections.synchronizedList()` or `Vector`
Multi-threaded (read-heavy)	`CopyOnWriteArrayList`

Stack-like behavior	Prefer `ArrayDeque` over `Stack`

Let me know if you want a complete working multithreaded example with both `ArrayList` and `Vector`.

4o