Java Vector Level 1

Vector

now we go level by level growth like a game on this java topic, provide coding example with its explanation, real-world use case example,neccessary hinds, points to remember,usual mistakes,etc.

start lvl1

Great! Let's start **Java `Vector`** from **Level 1** and grow level by level like a game. 🚀

Java Vector - Level 1

1️⃣ What is a Vector?

- **`Vector`** is a **legacy class** in Java (**`java.util.Vector`**) that implements a **dynamic array**.
- Similar to **`ArrayList`** but **synchronized** (thread-safe).
- Can grow or shrink automatically as elements are added or removed.
- Stores **objects only** (no primitives directly, need wrappers like **`Integer`**, **`Double`**, etc.).

2️⃣ Basic Syntax

```
java

import java.util.Vector;

public class VectorLevel1 {
    public static void main(String[] args) {
        // Create a vector
        Vector<String> fruits = new Vector<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");

        // Display vector
        System.out.println("Fruits Vector: " + fruits);

        // Get element by index
        System.out.println("First fruit: " + fruits.get(0));

        // Remove element
        fruits.remove("Banana");
        System.out.println("After removing Banana: " + fruits);

        // Check size
        System.out.println("Size of vector: " + fruits.size());
    }
}
```

Output:

```yaml
Fruits Vector: [Apple, Banana, Mango]
First fruit: Apple
After removing Banana: [Apple, Mango]
Size of vector: 2
```

3️⃣ Key Points to Remember

- Vectors are **synchronized** → safe for **multithreaded use**.
- Automatically **resizes** when capacity is exceeded.
- Stores **only objects**, not primitives.
- Elements can be accessed using **index**.
- **Legacy**: Modern code often prefers `ArrayList` unless you need synchronization.

4️⃣ Real-World Use Case

- Imagine a **chat application** where multiple threads add messages to a list.
  - `Vector<String> messages = new Vector<>();` ensures thread safety when multiple threads append messages.

5️⃣ Hints / Tips

- Use `add()` to insert at the end.
- Use `add(index, element)` to insert at a specific position.
- Use `remove(index)` or `remove(object)` to delete elements.
- `get(index)` fetches the element.
- `size()` returns the number of elements.
- `isEmpty()` checks if vector is empty.

6️⃣ Usual Mistakes

- Trying to store primitive types directly (`int`, `double`) instead of using wrappers (`Integer`, `Double`).
- Confusing `Vector` with `ArrayList` in **single-threaded** applications (ArrayList is faster).
- Assuming `Vector` methods are **unsynchronized**.

Level 1 ✅ — You now know **what a Vector is**, **how to create it**, **basic operations**, and **thread-safe use cases**.

If you're ready, we can move to **Level 2**, where we explore **advanced operations like iteration, capacity, and more methods**.

Do you want me to proceed to **Level 2**?

next

Awesome! Let's level up. 🎮

Java Vector – Level 2

1️⃣ Advanced Operations in Vector

Now that you know the basics, Level 2 focuses on:

- Iterating through a Vector
- Checking capacity and size
- Using some common useful methods

2️⃣ Coding Example

```java
import java.util.Vector;
import java.util.Enumeration;

public class VectorLevel2 {
    public static void main(String[] args) {
        Vector<String> fruits = new Vector<>();

        // Add elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Mango");
        fruits.add("Orange");

        // 1. Iteration using for loop
        System.out.println("Using for loop:");
        for(int i = 0; i < fruits.size(); i++){
            System.out.println(fruits.get(i));
        }

        // 2. Iteration using enhanced for loop
        System.out.println("\nUsing enhanced for loop:");
        for(String fruit : fruits){
            System.out.println(fruit);
        }

        // 3. Iteration using Enumeration (legacy style)
        System.out.println("\nUsing Enumeration:");
        Enumeration<String> e = fruits.elements();
        while(e.hasMoreElements()){
            System.out.println(e.nextElement());
        }

        // 4. Capacity and size
        System.out.println("\nCurrent size: " + fruits.size());
```

```
        System.out.println("Current capacity: " + fruits.capacity());

        // 5. Check if contains element
        System.out.println("\nContains Mango? " + fruits.contains("Mango"));

        // 6. Remove all elements
        fruits.clear();
        System.out.println("After clearing: " + fruits);
    }
}
```

Output:

```
sql


Using for loop:
Apple
Banana
Mango
Orange

Using enhanced for loop:
Apple
Banana
Mango
Orange

Using Enumeration:
Apple
Banana
Mango
Orange

Current size: 4
Current capacity: 10

Contains Mango? true
After clearing: []
```

3️⃣ Key Points

- **`size()`** → number of elements
- **`capacity()`** → how many elements Vector can hold **before resizing**
- **`Enumeration`** → old-school iteration (legacy)
- **`contains(obj)`** → check if element exists
- **`clear()`** → remove all elements

4️⃣ Real-World Use Case

- **History of commands in an application**:
  - Vector stores commands entered by users.
  - Threads might add/remove commands concurrently.
  - Capacity helps to know the allocated memory vs actual usage.

5️⃣ Hints / Tips

- `capacity()` doubles the current size if exceeded (default 10 initial capacity).
- Use `Enumeration` only when maintaining **legacy code**.
- Prefer **Iterator or enhanced for loop** in modern Java.

6️⃣ Usual Mistakes

- Confusing `size()` and `capacity()` — `capacity()` is **not the number of elements**.
- Using `Enumeration` but forgetting `hasMoreElements()`.
- Clearing the vector unintentionally with `clear()` in the middle of operations.

Level 2 ✅ — You now know **iteration methods, capacity, checking elements, and clearing**.

Next, **Level 3** can cover **thread-safe operations, synchronized access, and Vector vs ArrayList comparison**.

Do you want to proceed to **Level 3**?

next

Level 3, here we go! 🎮

Java Vector – Level 3: Thread-Safety & Advanced Usage

1️⃣ Synchronized / Thread-Safe Operations

- `Vector` methods like `add()`, `remove()`, `get()` are **synchronized** by default.
- This makes `Vector` **thread-safe**, unlike `ArrayList`.
- Useful in **multi-threaded applications** where multiple threads access the same collection.

2️⃣ Coding Example: Thread-Safe Vector

```
java

import java.util.Vector;

class Producer extends Thread {
    private Vector<Integer> sharedVector;

    Producer(Vector<Integer> v) {
        this.sharedVector = v;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            sharedVector.add(i);
            System.out.println("Producer added: " + i);
```

```
            try { Thread.sleep(100); } catch(Exception e) {}
        }
    }
}

class Consumer extends Thread {
    private Vector<Integer> sharedVector;

    Consumer(Vector<Integer> v) {
        this.sharedVector = v;
    }

    public void run() {
        for (int i = 1; i <= 5; i++) {
            if (!sharedVector.isEmpty()) {
                int val = sharedVector.remove(0);
                System.out.println("Consumer removed: " + val);
            }
            try { Thread.sleep(150); } catch(Exception e) {}
        }
    }
}

public class VectorLevel3 {
    public static void main(String[] args) {
        Vector<Integer> sharedVector = new Vector<>();

        Producer producer = new Producer(sharedVector);
        Consumer consumer = new Consumer(sharedVector);

        producer.start();
        consumer.start();
    }
}
```

Output (sample, may vary):

```
yaml


Producer added: 1
Consumer removed: 1
Producer added: 2
Producer added: 3
Consumer removed: 2
Producer added: 4
Consumer removed: 3
Producer added: 5
Consumer removed: 4
Consumer removed: 5
```

Explanation:

- Even though **Producer** and **Consumer** run concurrently, `Vector` ensures **thread-safety** for `add()` and `remove()`.
- No data corruption occurs.

3️⃣ Vector vs ArrayList

Feature	Vector	ArrayList
Synchronization	Synchronized	Not synchronized
Legacy	Yes	No
Performance	Slower (due to sync)	Faster

Feature	Vector	ArrayList
Use-case	Multi-threaded	Single-threaded

Hint: In modern applications, you usually use `ArrayList` + `Collections.synchronizedList()` if you need thread-safety.

4️⃣ Useful Methods in Level 3

- `addAll(Collection c)` → add all elements from another collection
- `removeAll(Collection c)` → remove all elements found in another collection
- `retainAll(Collection c)` → keep only elements present in another collection
- `indexOf(obj)` → first index of element
- `lastIndexOf(obj)` → last occurrence index

5️⃣ Real-World Use Case

- **Ticket booking system**: multiple threads updating seat availability in a shared list.
  - `Vector` ensures seats are booked correctly without conflicts.

6️⃣ Hints / Tips

- `Vector` is slower than `ArrayList` → use only when **thread-safety is needed**.
- Avoid using `Enumeration` in new code — prefer **Iterator**.
- `Vector` allows **null elements**.

7️⃣ Common Mistakes

- Assuming `Vector` is 100% safe for complex operations (like iterating + modifying at the same time) — still need external synchronization for compound actions.
- Confusing `synchronized` with **immutable** — elements inside Vector can still be changed.

Level 3 ✅ — You now know **thread-safe usage, Vector vs ArrayList, advanced methods, and real-world multi-threaded application**.

If you want, **Level 4** can cover **legacy methods, enumeration, listIterator, subList, and performance tips** for full mastery.

Do you want to move to **Level 4**?