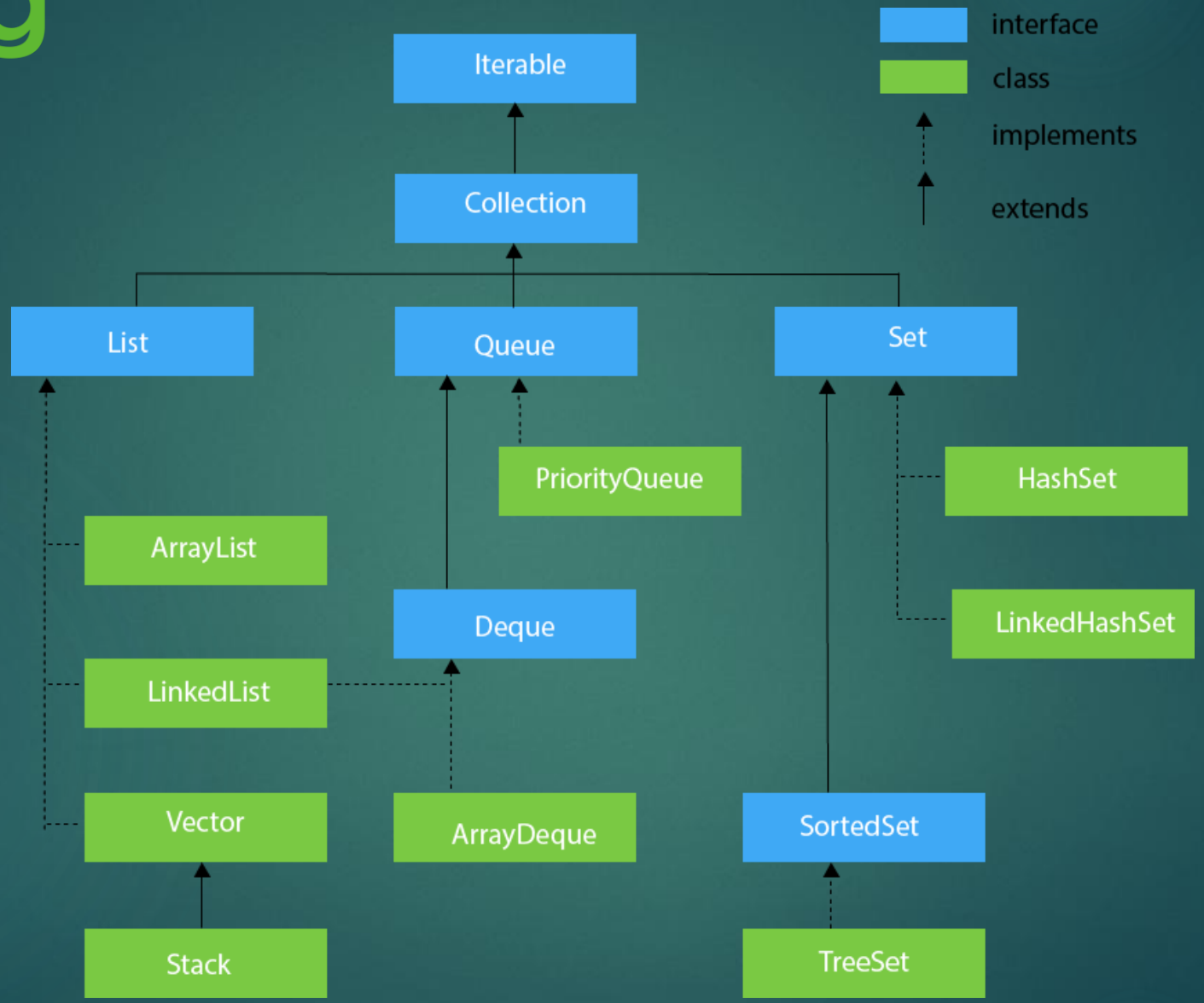

By Eslam Khder

What is a Vector in Java?

A `Vector` is a **class** in Java that stores a group of elements (objects), similar to an `ArrayList`.

It's part of the **Collection Framework**, and it can hold data like numbers, strings, or any kind of objects.

Package:

The `Vector` class is located in:

java Copy code

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

How it works:

A `Vector` works almost the same as an `ArrayList`, but there are two main differences:

1. It is **synchronized**, meaning it is **thread-safe** (safe to use in multi-threaded environments).
2. When it becomes full, it **automatically doubles its capacity** instead of creating a new array manually.

By Eslam Khder

VS Difference between `Vector` and `ArrayList`:

Feature	Vector	ArrayList
Synchronization	Synchronized (Thread-safe)	Not synchronized
Performance	Slower (because of synchronization)	Faster
Capacity Growth	Grows by 100% (doubles)	Grows by 50%

By Eslam Khder

What is a PriorityQueue (Simple Idea)

A PriorityQueue in Java is a special type of queue
where each element has a priority,
and the element with the highest priority is always served first,
not necessarily the one that was added first.

By Eslam Khder

Quick Example

java

Copy code

```java
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> queue = new PriorityQueue<>();

        queue.add(30);
        queue.add(10);
        queue.add(20);

        System.out.println(queue.poll()); // 10 → smallest has highest priority
        System.out.println(queue.poll()); // 20
        System.out.println(queue.poll()); // 30
    }
}
```

By Eslam Khder

⚙️ You can also define your own priority

For example, if you want **bigger numbers** to have higher priority:

java

Copy code

```java
PriorityQueue<Integer> queue = new PriorityQueue<>(Comparator.reverseOrder());
```

🎯 In simple words:

> A `PriorityQueue` is like a **smart queue** that decides who goes first **based on importance**, not based on who came first.

By Eslam Khder

What is an `ArrayDeque`?

`ArrayDeque` stands for:

> **Array Double-Ended Queue**

That means it's a **queue** where you can **add** and **remove** elements from **both ends** — from the **front** and from the **back**.

So, it can act as either:

- a **Queue** → first in, first out (**FIFO**)
- or a **Stack** → last in, first out (**LIFO**)

By Eslam Khder

Using it as a Stack

java Copy code

```java
ArrayDeque<Integer> stack = new ArrayDeque<>();

stack.push(10);
stack.push(20);
stack.push(30);

System.out.println(stack.pop()); // 30 → last in, first out
```

Using it as a Queue

java Copy code

```java
ArrayDeque<Integer> queue = new ArrayDeque<>();

queue.offer(10);
queue.offer(20);
queue.offer(30);

System.out.println(queue.poll()); // 10 → first in, first out
```

By Eslam Khder

1 What is a Map?

A **Map** is a **collection of key → value pairs.**

- **Key** = unique identifier
- **Value** = data for that key

Example:

java Copy code

```java
Map<String, Integer> map = new HashMap<>();
map.put("Eslam", 25);
map.put("Ahmed", 30);
map.put("Sara", 19);
```

No duplicate keys are allowed. If you add the same key again, the old value **replaces** the new one.

By Eslam Khder

2 How HashMap stores values

Step 1: Buckets (Array)

Imagine an **array of boxes** (buckets) where the data will be stored:

```
css                                                    Copy code

[0] [1] [2] [3] [4] [5] [6] [7]
```

Each **bucket** can store one or more key-value pairs.

Step 2: Hash function

When you insert a key, the map calculates a **hash** to decide which bucket to store it in:

```
bash                                                   Copy code

hash("Eslam") → 3
hash("Ahmed") → 5
hash("Sara")  → 7
```

By Eslam Khder

Step 3: Store in bucket

The key-value pair goes into the bucket at that index:

nginx

Copy code

```
Bucket 3: ("Eslam", 25)
Bucket 5: ("Ahmed", 30)
Bucket 7: ("Sara", 19)
```

> ✅ No duplicates allowed: If you try `map.put("Eslam", 40)`, it **replaces** 25 with 40.

By Eslam Khder

3 How operations work O(1)

Add/Update:

java Copy code

```
map.put("Ali", 22); // computes hash, finds bucket, stores it
```

Get value:

java Copy code

```
map.get("Ahmed"); // computes hash → bucket 5 → returns 30
```

Remove:

java Copy code

```
map.remove("Sara"); // goes to bucket 7 → removes the pair
```

Why O(1)?

- You **compute the bucket directly** using the hash → no need to check all elements.
- Access is **direct**, very fast.

3 How operations work O(1)

Add/Update:

java Copy code

```
map.put("Ali", 22); // computes hash, finds bucket, stores it
```

Get value:

java Copy code

```
map.get("Ahmed"); // computes hash → bucket 5 → returns 30
```

Remove:

java Copy code

```
map.remove("Sara"); // goes to bucket 7 → removes the pair
```

Why O(1)?

- You **compute the bucket directly** using the hash → no need to check all elements.
- Access is **direct**, very fast.

1 HashMap

- **Type:** Non-synchronized → **not thread-safe**
- **Allows null:** Yes, **key can be null, value can be null**
- **Order:** Not guaranteed → elements **may not appear in insertion order**
- **Performance:** Very fast because no synchronization
- **Use case:** When thread-safety is **not required**

Example:

java

Copy code

```java
HashMap<String, Integer> map = new HashMap<>();
map.put("Eslam", 25);
map.put("Ahmed", 30);
System.out.println(map); // Order may differ from insertion order
```

By Eslam Khder

2 Hashtable

- **Type:** Synchronized → **thread-safe**
- **Allows null:** No, **neither key nor value can be null**
- **Order:** Not guaranteed → like HashMap
- **Performance:** Slower than HashMap due to synchronization
- **Use case:** When you need **thread-safe operations**

Example:

java

Copy code

```java
Hashtable<String, Integer> table = new Hashtable<>();
table.put("Eslam", 25);
table.put("Ahmed", 30);
```

By Eslam Khder

3 LinkedHashMap

- **Type:** Like HashMap, **non-synchronized**
- **Allows null:** Yes, key and value can be null
- **Order:** **Maintains insertion order**
- **Extra feature:** Can maintain **access-order** (order changes when accessed)
- **Use case:** When you need to **preserve order**

Example:

java

Copy code

```java
LinkedHashMap<String, Integer> linkedMap = new LinkedHashMap<>();
linkedMap.put("Eslam", 25);
linkedMap.put("Ahmed", 30);
linkedMap.put("Sara", 19);
System.out.println(linkedMap);
// Output: {Eslam=25, Ahmed=30, Sara=19}
```

By Eslam Khder

🔹 Quick Comparison

Type	Synchronized	Allows null	Maintains order	Performance
HashMap	No	Yes	No	Very fast
Hashtable	Yes	No	No	Slower
LinkedHashMap	No	Yes	Yes (insertion order)	Fast

spring®

EraaSoft

1 What is a Set?

A Set is a collection that stores unique elements only.

- No duplicates – each element appears only once
- Order usually doesn't matter
- Ideal when you just need a collection of unique items

By Eslam Khder

2 HashSet

- **Type:** Unordered set
- **Allows null:** Yes
- **Order:** Not guaranteed → elements can appear in any order
- **Performance:** Very fast → add, remove, and contains are **O(1) on average**

Example:

java

Copy code

```java
HashSet<String> hashSet = new HashSet<>();
hashSet.add("Eslam");
hashSet.add("Ahmed");
hashSet.add("Sara");
System.out.println(hashSet); // Order not guaranteed
```

3 LinkedHashSet

- **Type:** Maintains **insertion order**
- **Allows null:** Yes
- **Order:** Elements appear in the **same order they were added**
- **Performance:** Almost as fast as HashSet, slightly slower due to order tracking

Example:

java

Copy code

```java
LinkedHashSet<String> linkedSet = new LinkedHashSet<>();
linkedSet.add("Eslam");
linkedSet.add("Ahmed");
linkedSet.add("Sara");
System.out.println(linkedSet); // Output: [Eslam, Ahmed, Sara]
```

Summary:

- Use HashSet if order doesn't matter
- Use LinkedHashSet if you want to preserve insertion order

By Eslam Khder

Set is built on HashMap

1 The truth: Set is built on HashMap

In Java, most **Set implementations** (like **HashSet**) **store their elements internally using a HashMap**.

Inside `HashSet`, you'll see something like:

java Copy code

```java
private transient HashMap<E,Object> map;
```

- Each element in the Set becomes a **key in the HashMap**.
- The value (`Object`) is **not important**, usually a dummy object:

java Copy code

```java
private static final Object PRESENT = new Object();
```

By Eslam Khder

Set is built on HashMap

2 Why use a HashMap?

The Set needs:

1. **Unique elements only** → HashMap ensures keys are unique
2. **Fast operations (O(1))** → HashMap gives fast add/remove/contains

So basically: **every element in the Set is a key in the HashMap**, and the value is just a placeholder.

---

3 Example

java Copy code

```java
HashSet<String> set = new HashSet<>();
set.add("Eslam");
set.add("Ahmed");
set.add("Sara");

// Internally, it's like having a HashMap:
map = {
    "Eslam"  -> PRESENT,
    "Ahmed"  -> PRESENT,
    "Sara"   -> PRESENT
}
```

- If you try to add `"Eslam"` again → HashMap prevents duplicates
- That's why **Set never allows duplicate elements**

By Eslam Khder

Set is built on HashMap

4️⃣ Summary

Point	Explanation
Why HashMap?	Ensures **unique keys** and fast operations
What is stored as value?	A dummy object like `PRESENT` because the value itself is not important
How duplicates are prevented?	HashMap does **not allow duplicate keys**, so every Set element is unique

By Eslam Khder