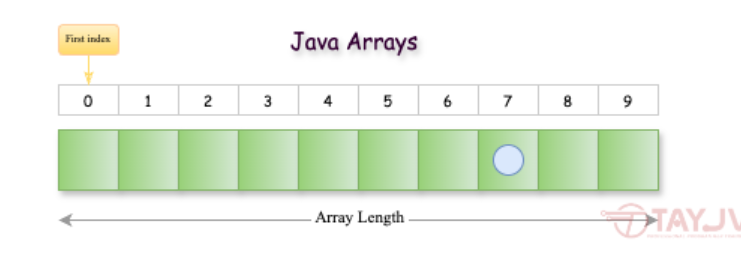
JAVA COLLECTION

1. Array



- Nêu hiểu biết về kiểu mảng trong java

+ **Array** trong Java là **cấu trúc dữ liệu tĩnh** để lưu trữ nhiều phần tử cùng kiểu.

+ Kích thước **cố định** khi khởi tạo → không thể thay đổi sau đó.

+ Các phần tử lưu liên tiếp trong bộ nhớ → truy cập nhanh bằng **chỉ số (index)**.

+ Index bắt đầu từ **0** → arr[0].

+ Có thể là mảng 1 chiều hoặc đa chiều (int[][] matrix).

+ Mảng là **object** trong Java → có thuộc tính length.

- Dùng kiểu mảng mang lại ưu , nhược điểm gì?

**Ưu điểm**

* **Truy cập nhanh**: O(1) khi biết index.
* **Đơn giản**: dễ dùng, dễ hiểu.
* Lưu trữ liên tục trong bộ nhớ → hiệu năng cao.
* Có thể dùng cho mảng primitive → tiết kiệm bộ nhớ hơn so với Collection.

**Nhược điểm**

* **Kích thước cố định**: phải biết trước số phần tử; không thể mở rộng.
* **Không có method tiện ích** (như add, remove, contains trong ArrayList).
* **Khó thao tác**: chèn/xóa phần tử phải dịch chuyển các phần tử khác → tốn chi phí O(n).
* **Không an toàn**: dễ lỗi ArrayIndexOutOfBoundsException nếu truy cập sai index.

👉 Vì vậy, trong thực tế:

* Dùng **Array** khi số phần tử cố định hoặc cần hiệu năng cực cao.
* Dùng **Collection (ArrayList, List, Set, Map, …)** khi dữ liệu động, nhiều thao tác thêm/xóa.

2. List Interface

- Nêu ra các đặc điểm **List Interface**

 Thuộc **Collection Framework** (java.util.List).

 Là **danh sách có thứ tự** (ordered collection).

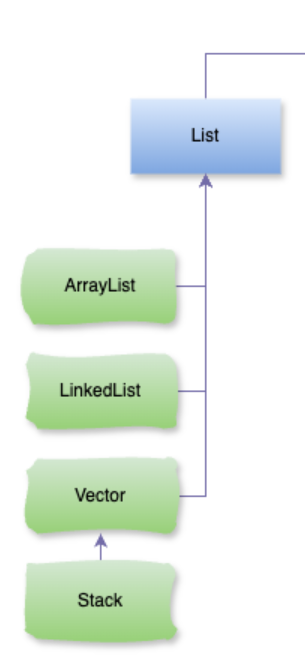
 Cho phép **các phần tử trùng lặp**.

 Có thể truy cập phần tử theo **index** (giống mảng).

 Cung cấp nhiều method tiện ích:

* add(E e), add(int index, E e)
* get(int index)
* remove(int index), remove(Object o)
* set(int index, E e)
* indexOf(Object o), lastIndexOf(Object o)
* subList(fromIndex, toIndex)

- Kể ra các **class**triển khai từ **List Interface**



Các class phổ biến:

* **ArrayList**
* **LinkedList**
* **Vector** (cũ, ít dùng)
* **Stack** (extends từ Vector)

- Phân biệt rõ trường hợp sử dụng của từng **class**đó

**+ ArrayList**

 Implemented bằng **mảng động**.

 Truy cập theo index rất nhanh (O(1)).

 Chèn/xóa **chậm** (O(n)) vì cần dịch chuyển phần tử.

 **Không đồng bộ (non synchronized)**.

 **Cho phép phần tử trùng lặp và giá trị null**

* Sử dụng ArrayList khi bạn cần một danh sách mà việc truy cập phần tử theo chỉ số là nhanh chóng và không cần phải thêm hoặc xóa phần tử quá nhiều.
* Nếu cần thao tác thêm/xóa phần tử thường xuyên ở đầu hoặc giữa danh sách, có thể cân nhắc sử dụng **LinkedList** thay thế.

**Các phương thức phổ biến của ArrayList**

* **add(E e)**: Thêm phần tử vào cuối danh sách.
* **add(int index, E element)**: Thêm phần tử vào vị trí chỉ định.
* **remove(Object o)**: Xóa phần tử đầu tiên có giá trị tương ứng trong danh sách.
* **remove(int index)**: Xóa phần tử tại vị trí chỉ định.
* **get(int index)**: Trả về phần tử tại vị trí chỉ định.
* **set(int index, E element)**: Cập nhật phần tử tại vị trí chỉ định với giá trị mới.
* **size()**: Trả về số lượng phần tử trong danh sách.
* **clear()**: Xóa toàn bộ các phần tử trong danh sách.
* **isEmpty()**: Kiểm tra xem danh sách có rỗng không.
* **contains(Object o)**: Kiểm tra xem danh sách có chứa phần tử cụ thể không.

**+ LinkedList**

 Implemented bằng **danh sách liên kết kép (doubly linked list)**.

 Truy cập theo index **chậm** (O(n)).

 Chèn/xóa ở đầu/cuối nhanh (O(1)).

 Có thể dùng như **Queue** hoặc **Deque**.

* **Các phương thức quan trọng**:
  + addFirst(), addLast(): Thêm phần tử vào đầu/cuối danh sách.
  + removeFirst(), removeLast(): Xóa phần tử đầu/cuối danh sách.
  + getFirst(), getLast(): Truy cập phần tử đầu/cuối.
  + add(index, element): Thêm phần tử vào vị trí chỉ định.
  + remove(index): Xóa phần tử tại vị trí chỉ định.

Dùng khi: nhiều thao tác **thêm/xóa ở đầu/cuối**, ít truy xuất ngẫu nhiên.

+**Vector**

* Giống ArrayList nhưng **synchronized** (thread-safe).
* Hiệu năng kém hơn do phải đồng bộ hóa.
* Hầu như không còn dùng trong code hiện đại, thay bằng ArrayList + Collections.synchronizedList() hoặc CopyOnWriteArrayList.

**Stack**

* Kế thừa từ Vector, tuân theo nguyên tắc **LIFO (Last In, First Out)**.
* Có method push(), pop(), peek().

=> Dùng khi cần mô phỏng stack.

3. Queue Interface

- Nêu các đặc điểm của **Queue Interface, Dequeue Interface**

**+ Đặc điểm của Queue Interface**

 Thuộc **java.util**.

 Là **cấu trúc dữ liệu hàng đợi** → thường theo nguyên tắc **FIFO (First In, First Out)**.

 Các thao tác chính:

* add(e) / offer(e) → thêm phần tử vào cuối hàng đợi.
* remove() / poll() → lấy và loại bỏ phần tử ở đầu hàng đợi.
* element() / peek() → chỉ lấy phần tử ở đầu mà không loại bỏ.

 **Khác biệt nhỏ**:

* add() & remove() ném exception khi thất bại.
* offer() & poll() trả về false/null khi thất bại → an toàn hơn.

**+ Đặc điểm của Deque Interface**

 **Deque (Double Ended Queue)** = hàng đợi 2 đầu.

 Có thể chèn/lấy phần tử ở **cả đầu và cuối**.

 Hỗ trợ cả:

* **FIFO** (queue bình thường).
* **LIFO** (giống stack).

 Các method chính:

* Thêm: addFirst(e), addLast(e)
* Xóa: removeFirst(), removeLast()
* Truy cập: getFirst(), getLast()

- Kể ra các **class**triển khai từ **Queue Interface, Dequeue Interface**, phân biệt trường hợp sử dụng tương

**✅ Queue**

* **LinkedList**
  + Cài đặt bằng danh sách liên kết kép.
  + Dùng tốt cho queue thường (FIFO).
* **PriorityQueue**
  + Hàng đợi ưu tiên → phần tử được sắp xếp theo **natural order** hoặc **Comparator**.
  + Không đảm bảo thứ tự FIFO.
  + Dùng trong thuật toán (Dijkstra, A\*…).
* **ArrayDeque** (cũng implement Queue)
  + Nhanh hơn LinkedList trong nhiều trường hợp.
  + Không có giới hạn kích thước (tự mở rộng).

**✅ Deque**

* **ArrayDeque**
  + Cấu trúc mảng động hai đầu.
  + Nhanh, không synchronized.
* **LinkedList**
  + Cũng implement Deque.
  + Thích hợp nếu cần nhiều chèn/xóa ở đầu/cuối.

**Khi nào nên sử dụng PriorityQueue?**

* **Cần xử lý các phần tử theo độ ưu tiên**:
  + Ví dụ: Một hàng đợi tác vụ trong hệ điều hành (các tác vụ quan trọng được xử lý trước).
* **Không quan tâm đến thứ tự chèn**:
  + PriorityQueue sắp xếp dựa trên độ ưu tiên, không dựa trên thứ tự chèn.
* **Hiệu suất tốt**:
  + Khi cần một cấu trúc dữ liệu gọn nhẹ và nhanh để xử lý các phần tử theo thứ tự ưu tiên.

**Khi nào nên sử dụng ArrayDeque**

* Khi bạn cần một deque hiệu suất cao mà không yêu cầu **thread-safety**.
* Khi cần thực hiện các thao tác chèn/xóa ở cả hai đầu nhanh hơn so với các cấu trúc khác như LinkedList.
* Khi bạn cần một cấu trúc dữ liệu làm stack hoặc queue:
  + Stack: Thay thế cho Stack class với các phương thức như push(), pop().
  + Queue: Làm queue với các phương thức offer(), poll().

4. Set Interface

- Nêu ra các đặc điểm **Set Interface**

* Thuộc java.util.
* Là **tập hợp không chứa phần tử trùng lặp**.
* **Không đảm bảo thứ tự** (tùy vào class triển khai).
* Một số implementation có thể cho phép phần tử null.
* Các method chính: add(), remove(), contains(), size(), iterator().

- Kể ra các **class**triển khai từ **Set Interface**

 **HashSet**

* Cài đặt bằng **HashMap** bên dưới.
* **Không đảm bảo thứ tự** các phần tử.
* Cho phép 1 giá trị null.
* Hiệu năng cao cho thao tác add, remove, contains (O(1)).

 **LinkedHashSet**

* Kế thừa từ HashSet nhưng **giữ nguyên thứ tự chèn**.
* Cho phép 1 giá trị null.
* Tốc độ hơi chậm hơn HashSet một chút.

 **TreeSet**

* Cài đặt bằng **TreeMap** (dựa trên cấu trúc cây đỏ-đen).
* Các phần tử được **sắp xếp tăng dần (natural order)** hoặc theo **Comparator**.
* **Không cho phép null** (sẽ ném NullPointerException).
* Thao tác O(log n).

- Phân biệt rõ trường hợp sử dụng của từng **class**đó

**HashSet**

* Khi chỉ cần **kiểm tra sự tồn tại** (unique) mà không quan tâm đến thứ tự.
* Ví dụ: danh sách email đăng ký, danh sách ID duy nhất.

**LinkedHashSet**

* Khi vừa muốn **loại bỏ trùng lặp**, vừa cần **giữ thứ tự chèn**.
* Ví dụ: lưu lịch sử duyệt web mà không trùng URL.

**TreeSet**

* Khi cần **dữ liệu luôn được sắp xếp** hoặc **tìm kiếm theo phạm vi**.
* Ví dụ: leaderboard, danh sách điểm số, tìm phần tử lớn hơn/nhỏ hơn một giá trị.

5. Map Interface

- Nêu ra các đặc điểm **Map Interface**

* Thuộc java.util.
* Lưu trữ dữ liệu theo **cặp key – value** (key → value).
* **Key là duy nhất**, nhưng **value có thể trùng lặp**.
* Truy cập phần tử thông qua **key** thay vì index.
* Các method chính:
  + put(key, value) → thêm hoặc cập nhật value theo key.
  + get(key) → lấy value theo key.
  + remove(key) → xóa theo key.
  + containsKey(key), containsValue(value) → kiểm tra tồn tại.
  + keySet(), values(), entrySet() → lấy tập key, value hoặc cả entry.

- Kể ra các **class**triển khai từ **Map Interface**

 **HashMap**

* Cài đặt bằng **Hash table**.
* Không đảm bảo thứ tự.
* Cho phép **1 key null** và nhiều value null.
* Hiệu năng cao (O(1) trung bình).

 **LinkedHashMap**

* Kế thừa HashMap, nhưng **giữ thứ tự chèn** (insertion order).
* Cho phép 1 key null, nhiều value null.
* Tốc độ chậm hơn HashMap một chút.

 **TreeMap**

* Cài đặt bằng **Red-Black Tree**.
* Các key được **sắp xếp tăng dần** (natural order hoặc theo Comparator).
* **Không cho phép key null**.
* Thao tác O(log n).

 **Hashtable** (cũ)

* Giống HashMap nhưng **synchronized** (thread-safe).
* **Không cho phép key hoặc value null**.
* Hiệu năng thấp hơn → ít dùng, thay bằng ConcurrentHashMap.

 **ConcurrentHashMap**

* Dùng trong môi trường **đa luồng**.
* Không khóa toàn bộ map như Hashtable, mà khóa theo segment → hiệu năng cao hơn.
* Không cho phép null key hoặc null value.

- Phân biệt rõ trường hợp sử dụng của từng **class**đó

**HashMap**

* Khi cần tra cứu key → value nhanh và không quan tâm thứ tự.
* Ví dụ: lưu thông tin user theo userId.

**LinkedHashMap**

* Khi cần vừa **tra cứu nhanh**, vừa **duy trì thứ tự chèn**.
* Dùng cho **cache LRU** (Least Recently Used).

**TreeMap**

* Khi cần dữ liệu **luôn được sắp xếp theo key**.
* Dùng cho **range query**, **leaderboard**, **dictionary**.

**Hashtable**

* Cũ, thread-safe nhưng chậm. Hiện nay ít dùng.

**ConcurrentHashMap**

* Khi cần **Map thread-safe** trong môi trường đa luồng, hiệu năng cao.
* Ví dụ: lưu session user trong server.