**[JAVA COLLECTION](https://classroom.google.com/w/Njc0NDQwMTUzOTQ4/tc/Njc0NDQ3MDQ0NzIz)**

**1. List**

# **1. Nêu ra các đặc điểm List Interface?**

### **1. Thứ tự lưu trữ (Order)**

* List duy trì thứ tự của các phần tử theo thứ tự chèn vào. Điều này có nghĩa là khi bạn thêm phần tử vào List, chúng sẽ được lưu trữ và duy trì theo thứ tự bạn thêm.

### **2. Cho phép phần tử trùng lặp (Duplicate Elements)**

* Một danh sách có thể chứa các phần tử trùng lặp. Nghĩa là bạn có thể thêm nhiều phần tử có giá trị giống nhau vào một danh sách.

### **3. Truy cập phần tử bằng chỉ số (Index-based Access)**

* List hỗ trợ truy cập phần tử theo chỉ số, nghĩa là bạn có thể truy xuất hoặc cập nhật phần tử dựa trên vị trí của nó trong danh sách.
* Phương thức get(int index) cho phép truy xuất phần tử dựa trên chỉ số.
* Phương thức set(int index, E element) cho phép thay đổi giá trị của phần tử tại vị trí chỉ định.

### **4. Kích thước linh hoạt (Resizable)**

* List có kích thước động, tức là bạn có thể thêm hoặc xóa phần tử mà không cần lo lắng về kích thước ban đầu (điều này khác với mảng).
* Các phương thức add(), remove(), clear() giúp bạn quản lý kích thước của danh sách một cách linh hoạt.

### **5. Các triển khai của List Interface**

List là một interface, và các lớp phổ biến triển khai List là:

* ArrayList: Lưu trữ các phần tử trong một mảng động, truy cập nhanh, nhưng chèn/xóa ở giữa chậm.
* LinkedList: Lưu trữ các phần tử trong một danh sách liên kết kép, chèn/xóa nhanh, nhưng truy cập chậm.
* Vector: Giống ArrayList nhưng được đồng bộ hóa (synchronized), thích hợp cho môi trường đa luồng.

### **6. Cho phép phần tử null**

* List có thể chứa giá trị null, nhưng bạn cần cẩn thận khi thao tác với các phần tử này vì việc truy cập phần tử null có thể gây ra lỗi NullPointerException.

# **2. Nêu ra các đặc điểm ArrayList?**

Kế thừa tất cả các đặc tính của List Collection.

### **1. Mảng động (Dynamic Array)**

* **ArrayList** được triển khai dựa trên một **mảng động** bên trong. Mảng động này có khả năng thay đổi kích thước khi cần thiết. Khi dung lượng của mảng hiện tại không đủ để chứa thêm phần tử, ArrayList sẽ tự động tạo một mảng mới với dung lượng lớn hơn, sao chép các phần tử từ mảng cũ sang mảng mới và loại bỏ mảng cũ.
* Ban đầu, mảng bên trong của ArrayList có một kích thước mặc định. Khi cần thêm phần tử vượt quá kích thước hiện tại, ArrayList sẽ tự động tăng kích thước mảng theo quy luật nhất định (thường là tăng kích thước lên gấp 1.5 lần so với dung lượng hiện tại).

### **2. Chỉ mục dựa trên mảng (Index-based Array)**

* Các phần tử trong ArrayList được lưu trữ liên tục trong bộ nhớ, và mỗi phần tử có thể truy cập được thông qua chỉ số (index), giống như mảng thông thường.
* Thao tác truy cập phần tử có thời gian truy cập trung bình là O(1), do có thể truy cập trực tiếp dựa trên chỉ số.

### 3. Phương thức thêm và xóa phần tử (Add and Remove Operations)

* Thao tác thêm một phần tử vào ArrayList có thời gian trung bình là O(1), vì phần tử thường được thêm vào cuối mảng. Tuy nhiên, nếu cần phải tăng kích thước mảng, chi phí cho thao tác này sẽ là O(n) do cần phải sao chép tất cả các phần tử từ mảng cũ sang mảng mới.
* Thao tác xóa phần tử tại một vị trí nhất định có thời gian O(n), vì sau khi xóa phần tử, tất cả các phần tử sau nó cần phải dịch chuyển để lấp vào khoảng trống.

# **3. Nêu ra các đặc điểm LinkedList?**

### **1. Cấu trúc dữ liệu danh sách liên kết (Linked List)**

* LinkedList trong Java được triển khai như một danh sách liên kết đôi (doubly linked list). Mỗi phần tử của LinkedList là một **node** chứa dữ liệu và con trỏ tới phần tử trước đó và phần tử kế tiếp.
* Không giống như ArrayList, các phần tử trong LinkedList không được lưu trữ liên tiếp trong bộ nhớ. Thay vào đó, mỗi phần tử là một đối tượng riêng biệt với một liên kết tới phần tử trước và sau nó.

### **2. Hiệu suất tốt hơn cho thao tác thêm/xóa (Better for Insertions/Deletions)**

* Do cấu trúc danh sách liên kết, việc thêm và xóa phần tử ở bất kỳ vị trí nào trong LinkedList đều có độ phức tạp thời gian là O(1), đặc biệt là ở đầu hoặc cuối danh sách. Bạn chỉ cần cập nhật các con trỏ của phần tử trước và sau mà không cần phải dịch chuyển phần tử như trong ArrayList.
* Tuy nhiên, nếu bạn muốn truy cập ngẫu nhiên (random access) một phần tử theo chỉ số (index), thời gian truy cập sẽ là O(n), vì LinkedList phải duyệt qua danh sách để tìm phần tử đó.

### **3. Không có truy cập chỉ mục nhanh (No Fast Random Access)**

* Trái ngược với ArrayList, nơi truy cập phần tử thông qua chỉ số rất nhanh (O(1)), trong LinkedList, truy cập phần tử theo chỉ số có độ phức tạp O(n). Điều này là do LinkedList phải duyệt qua các phần tử từ đầu hoặc cuối danh sách để tìm phần tử cần thiết.

### **4. Sử dụng nhiều bộ nhớ hơn (More Memory Overhead)**

* LinkedList sử dụng nhiều bộ nhớ hơn ArrayList vì mỗi phần tử trong danh sách phải lưu trữ thêm hai con trỏ (một tới phần tử trước đó và một tới phần tử tiếp theo). Điều này khiến LinkedList kém hiệu quả hơn về bộ nhớ so với ArrayList, nhất là khi có nhiều phần tử nhỏ.

### **5. Tính linh hoạt của cấu trúc dữ liệu (Deque Operations)**

* LinkedList triển khai giao diện Deque, cho phép nó hoạt động như một **queue** (hàng đợi), **deque** (hàng đợi hai đầu) hoặc **stack** (ngăn xếp). Điều này mang lại tính linh hoạt cao trong việc thêm, xóa và truy cập phần tử từ cả hai đầu danh sách (đầu hoặc cuối).
* Các phương thức như addFirst(), addLast(), removeFirst(), removeLast(), getFirst(), và getLast() rất hữu ích khi làm việc với hàng đợi và ngăn xếp.

### **7. Thích hợp với các hoạt động thêm/xóa thường xuyên (Ideal for Frequent Insertions/Deletions)**

* Do hiệu suất tốt khi thêm và xóa, LinkedList là lựa chọn lý tưởng cho các ứng dụng yêu cầu thêm/xóa phần tử thường xuyên, đặc biệt là khi thêm hoặc xóa ở đầu và cuối danh sách.

### **8. Không duy trì kích thước cố định (No Fixed Size)**

* Giống như ArrayList, LinkedList có thể thay đổi kích thước động khi các phần tử được thêm hoặc xóa. Điều này cho phép linh hoạt trong việc quản lý bộ nhớ mà không cần biết trước số lượng phần tử.

### **9. Duy trì thứ tự chèn (Maintains Insertion Order)**

* LinkedList duy trì thứ tự của các phần tử theo thứ tự mà chúng được thêm vào danh sách. Điều này có nghĩa là khi bạn duyệt qua LinkedList, các phần tử sẽ được trả về theo đúng thứ tự chúng đã được thêm vào.

# **4. Nêu ra các đặc điểm Vector?**

### **1. Kế thừa từ List và triển khai giao diện List**

* Vector kế thừa từ lớp **AbstractList** và triển khai các giao diện **List**, **RandomAccess**, **Cloneable**, và **Serializable**. Điều này có nghĩa là Vector hỗ trợ tất cả các thao tác mà một danh sách (list) cần có, như thêm, xóa, truy cập và duyệt qua các phần tử.

### **2. Đồng bộ hóa (Synchronized)**

* Điểm khác biệt lớn nhất giữa Vector và ArrayList là Vector được **đồng bộ hóa** (synchronized) mặc định. Điều này có nghĩa là Vector an toàn khi sử dụng trong các môi trường **đa luồng**. Các phương thức của Vector được đồng bộ hóa để đảm bảo rằng chỉ một luồng có thể thao tác trên đối tượng tại một thời điểm.
* Tuy nhiên, việc đồng bộ hóa cũng làm giảm hiệu suất của Vector so với các lớp không đồng bộ như ArrayList, đặc biệt trong môi trường đơn luồng (single-threaded).

### **3. Kích thước động (Dynamic Size)**

* Giống như ArrayList, Vector có thể thay đổi kích thước linh hoạt khi thêm hoặc xóa phần tử. Khi dung lượng ban đầu bị đầy, Vector sẽ tự động tăng kích thước để chứa thêm phần tử. Tuy nhiên, khi tăng dung lượng, Vector tăng **gấp đôi** kích thước hiện tại, trong khi ArrayList chỉ tăng thêm **50%**.
* Điều này có thể gây ra lãng phí bộ nhớ nếu bạn không cần nhiều phần tử.

### **4. Truy cập ngẫu nhiên nhanh (Fast Random Access)**

* Vector lưu trữ các phần tử trong một mảng nội bộ, giống như ArrayList. Điều này giúp cho việc truy cập phần tử theo chỉ số (index) rất nhanh, với độ phức tạp thời gian là **O(1)**. Bạn có thể truy xuất các phần tử trong Vector thông qua chỉ số tương tự như cách làm với mảng.

### **5. Thêm và xóa phần tử (Insertions and Deletions)**

* Do Vector sử dụng mảng nội bộ để lưu trữ phần tử, việc thêm hoặc xóa phần tử ở giữa danh sách có thể gây ra độ phức tạp **O(n)** do cần phải dịch chuyển các phần tử. Việc thêm phần tử ở cuối danh sách có độ phức tạp trung bình là **O(1)** nếu còn đủ dung lượng trong mảng.

### **6. Hỗ trợ lặp qua phần tử (Iteration Support)**

* Bạn có thể lặp qua các phần tử trong Vector bằng cách sử dụng các vòng lặp như **for-each**, **for-loop**, hoặc **Iterator**. Tuy nhiên, nếu bạn muốn lặp qua Vector trong môi trường đa luồng, bạn cần sử dụng phiên bản đồng bộ của **Iterator**, đó là Enumeration.

### **7. Hỗ trợ sao chép nông (Shallow Copy)**

* Vector hỗ trợ sao chép nông (shallow copy) thông qua phương thức clone(). Điều này có nghĩa là nó tạo ra một đối tượng Vector mới với các tham chiếu tới cùng các phần tử trong danh sách gốc.

### **8. Duy trì thứ tự chèn (Maintains Insertion Order)**

* Vector duy trì thứ tự của các phần tử theo thứ tự mà chúng được thêm vào danh sách. Khi duyệt qua Vector, các phần tử sẽ được trả về theo đúng thứ tự mà chúng đã được chèn vào.

### **9. Không hỗ trợ kiểu dữ liệu nguyên thủy (No Primitive Types)**

* Giống như các lớp khác trong Java Collections Framework, Vector không hỗ trợ lưu trữ trực tiếp các kiểu dữ liệu nguyên thủy (primitive types) như int, char, double. Bạn cần sử dụng các lớp đối tượng bọc như Integer, Character, Double.

### **10. Phù hợp cho môi trường đa luồng (Thread-Safe but Slower)**

* Vector là một lựa chọn tốt trong các môi trường yêu cầu tính an toàn trong đa luồng (thread-safe), vì các phương thức của nó đã được đồng bộ hóa. Tuy nhiên, nếu không cần đồng bộ hóa, việc sử dụng ArrayList sẽ nhanh hơn vì không có chi phí đồng bộ hóa.

### **11. Tăng kích thước linh hoạt (Capacity Increment)**

* Vector có thể tùy chỉnh cách tăng kích thước thông qua **capacity increment**. Điều này cho phép bạn kiểm soát số lượng phần tử sẽ được thêm vào mảng nội bộ khi cần mở rộng. Nếu không chỉ định, Vector sẽ tăng kích thước gấp đôi dung lượng hiện tại.