Java # Collections

1. Objectives
   1. Understandings Java’s Collections.
   2. Using ArrayList and Comparator.
   3. Understand about LinkedList, Stack, Queue, TreeSet, HashSet and HashMap.
2. Research

**SPRING 2022**

**PRO192 – Object-Oriented Programming**

**SE1606**

**Group 4**

**Member:**

1. Võ Quốc Thái CE160568
2. Phạm Đức Duy CE160896
3. Ngô Hải Bằng CE160624
4. Quách Đăng Khoa CE160561
5. Trần Tuấn Kiệt CE161026

**Question**

1. **What is ArrayList? How to manage data (insert, delete, search, sort) using ArrayList? What is ArrayList used for?**

Answer: <CE160568> <Võ Quốc Thái >

ArrayList là một lớp kế thừa lớp AbstractList và triển khai của List Interface trong Collections Framework nên nó sẽ có một vài đặc điểm và phương thức tương đồng với List. ArrayList được sử dụng như một mảng động để lưu trữ các phần tử.

Management data:

ArrayList<String> list = **new** ArrayList<String>();

Insert:

List.set(int index, Object o);

Delete

List.clear(); // xóa tất cả các phần tử

List.remove(int index); // xóa theo chỉ mục

List.remove(Object obj); // xóa theo đối tượng

Search:

List.lastIndexOf(Object o); // tìm vị trí xuất hiện cuối cùng

List.indexOf(Object o); // tìm vị trí xuất hiện đầu tiên

List.contain(Object o) // tìm kiếm trực tiếp

Sort:

Collections.sort(list);

ArrayList trong Java được sử dụng để lưu trữ bộ sưu tập các phần tử có kích thước linh hoạt. Trái với các mảng được cố định về kích thước, một ArrayList sẽ tự động tăng kích thước của nó khi các phần tử mới được thêm vào nó.

1. **What is LinkedList? How to manage data using LinkedList? What are pro and cons of LinkedList?**

Answer: <CE160568> <Võ Quốc Thái >

LinkedList là một lớp kế thừa lớp AbstractSequentialList và triển khai của List, Queue Interface trong Collections Framework nên nó sẽ có một vài đặc điểm và phương thức tương đồng với List, Queue. Lớp LinkedList trong java sử dụng cấu trúc danh sách liên kết kép Doubly để lưu trữ các phần tử.

Management data

LinkedList<String> list = **new** LinkedList<String>();

Insert:

List.set(int index, Object o);

Delete:

List.clear(); // xóa tất cả các phần tử

List.remove(int index); // xóa theo chỉ mục

List.remove(Object obj); // xóa theo đối tượng

Search:

List.lastIndexOf(Object o); // tìm vị trí xuất hiện cuối cùng

List.indexOf(Object o); // tìm vị trí xuất hiện đầu tiên

List.contain(Object o) // tìm kiếm trực tiếp

Sort:

Collections.sort(list);

Ưu điểm:

-Tiết kiếm bộ nhớ và cấp phát động: Không như array cần 1 lượng chỉ định ô nhớ trên bộ nhớ ngay khi khỏi tạo. Linked list chỉ sử dụng bộ nhớ để lưu trữ khi dữ liệu thực sự được lưu vào linked list.

-Nó còn có thể lưu các phần tử ở bất cứ đâu được phép trên bộ nhớ mà không cần các ô nhớ liền kề nhau như array

-Quick insertion (Thêm rất nhanh với complexity chỉ là O(1))

-Quick deletion (Xóa nhanh)

Nhược điểm:

Tìm kiểm chậm do phải duyệt qua nhiều node để đến được node cần tìm

1. **What are different between static array (int[] a) and ArrayList and LinkedList?**

Answer: <CE160568> <Võ Quốc Thái >

|  |  |
| --- | --- |
| Array | ArrayList và LinkedList |
| Kích thước cố định | Kích thước thay đổi được |
| Có thể lưu trữ kiểu nguyên thủy và đối tương | Chỉ có thể lưu trữ đối tượng |
| Tốc độ lưu trữ và thao tác nhanh hơn | Tốc độ lưu trữ và thao tác chậm |
| Chỉ có phương thức length | Có nhiều phương thức thao tác với dữ liệu |

1. **What is Stack? How to manage data using Stack? What is Stack used for?**

Answer: <CE161026><Trần Tuấn Kiệt>

Sử dụng Stack để lưu trữ các phần tử theo dạng ngăn xếp. Anh lấy ví dụ như mình sắp xếp một chồng dĩa gồm 10 cái dĩa. Thì cái dĩa nào được xếp đầu tiên sẽ ở phía dưới cùng, và cái dĩa sắp xếp sau cùng sẽ ở phía trên cùng. Như vậy khi ta lấy từng cái dĩa thì ta sẽ lấy cái trên cùng trước. Các phần tử của Stack được hoạt động theo nguyên lý phần tử nào được thêm sau cùng sẽ được lấy ra đầu tiên. Nó khác với Queue là phần tử nào được thêm đầu tiên sẽ được lấy ra đầu tiên.

1. Tạo một Stack

Stack stack = new Stack();

2. Thêm một phần tử vào Stack

Stack<String> stack = new Stack<String>();

stack.push("1");

3. Lấy một phần tử ra khỏi Stack

Stack<String> stack = new Stack<String>();

stack.push("1");

String topElement = stack.pop();

5. Tìm kiếm trong Stack

Stack<String> stack = new Stack<String>();

stack.push("1");

stack.push("2");

stack.push("3");

int index = stack.search("3"); //index = 3

6. Kích thước của Stack

Stack<String> stack = new Stack<String>();

stack.push("1");

stack.push("2");

stack.push("3");

int size = stack.size();

7. Duyệt qua các phần tử trong Stack

Stack<String> stack = new Stack<String>();

stack.push("123");

stack.push("456");

stack.push("789");

Iterator iterator = stack.iterator();

while(iterator.hasNext()){

Object value = iterator.next();

}

Và các công dụng cơ bản của Stack:

1. boolean empty()

Kiểm tra nếu Stack này là trống. Trả về true nếu nó trống và false nếu stack chứa các phần tử

2. Object peek( )

Trả về phần tử trên cùng của Stack, nhưng không gỡ bỏ nó

3. Object pop( )

Trả về phần tử trên cùng của Stack, gỡ bỏ nó

4. Object push(Object element)

Đẩy phần tử lên trên cùng của Stack. Cũng trả về phần tử đó

5.int search(Object element)

Tìm kiếm phần tử trong Stack. Nếu tìm thấy, offset của nó từ trên cùng của Stack được trả về. Nếu không, nó trả về -1

1. **What is Queue? How to manage data using Queue? What is Queue used for?**

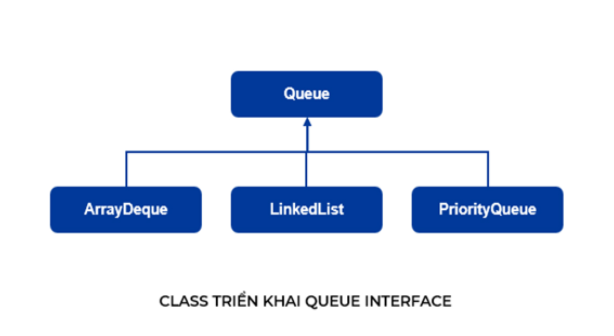
Answer: < CE161026> <Trần Tuấn Kiệt>

*Trong lập trình Java, tập hợp Queue được sử dụng để lưu trữ các phần tử theo một thứ tự, nó cũng tương tự như List nhưng cơ chế hoạt động của Queue là First In First Out. Có nghĩa là phần tử nào được add vào đầu tiên thì khi lấy ra thì được lấy ra đầu tiên. Các em hãy tưởng tượng như khi mình đi siêu thị và xếp hàng chờ thanh toán. Ai xếp hàng trước thì sẽ được thanh toán trước. Ai ở sau cùng sẽ thanh toán sau.*

**1.1. CÁC CLASS TRIỂN KHAI QUEUE INTERFACE**

Vì **Queue** là một **interface**, nên chúng ta không thể triển khai trực tiếp interface đó.  
  
  
Để sử dụng các tính năng của cấu trúc dữ liệu Queue, chúng ta cần sử dụng các class triển khai nó:

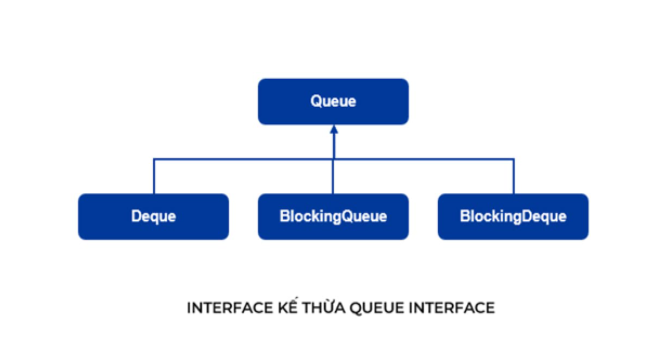
* ArrayDeque
* LinkedList
* PriorityQueue



**1.2. CÁC INTERFACE KẾ THỪA QUEUE**

Trong Java, Queue interface cũng được kế thừa bởi nhiều subinterface khác nhau:

* Deque
* BlockingQueue
* BlockingDeque



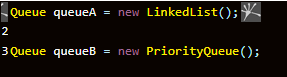
**Queue** có nghĩa là "**Hàng đợi**" (bạn có thể liên tưởng đến việc xếp hàng vậy)  
  
  
Trong hàng đợi (trong Queue), các phần tử được lưu trữ và truy cập theo cách nhập trước, xuất trước.  
  
  
Có nghĩa là:

* Khi thêm phần tử thì sẽ xếp vào phía sau
* Khi loại bỏ phần tử thì sẽ loại bỏ từ phía trước



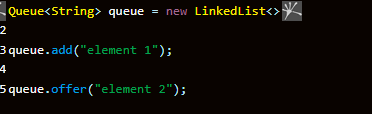
**1.3 Chúng ta có thể sử dụng các tập hợp của Queue như sau**

* Linkedlist
* PriorityQueue



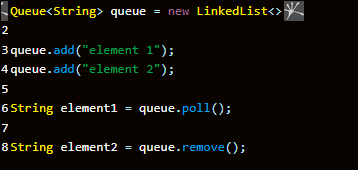
## **1.4 Thêm một phần tử vào Queue**

Chúng ta có 2 cách để thêm giá trị vào Queue. Các phần tử sẽ được add vào cuối của Queue, chúng ta có thể sử dụng phương thức add hoặc offer để add các phần tử vào cuối. Sự khác nhau giữa add và offer là khi Queue đã đầy nếu ta sử dụng phương thức add thì sẽ nhận được ngoại lệ (Exception) còn offer thì chúng ta nhận được giá trị false.



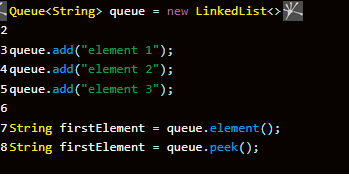
## **1.5 Lấy một phần tử từ Queue**

Chúng ta có thể sử dụng phương thức poll và remove để lấy phần tử đầu tiên của Queue. Sự khác nhau giữa poll và remove là nếu Queue là rỗng thì phương thức remove sẽ ném lỗi còn poll sẽ trả về giá trị null.



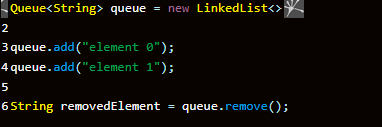
## **1.6 Lấy một phần tử đầu tiên từ Queue**

Để lấy phần tử đầu tiên của Queue chúng ta có thể sử dụng phương thức element hoặc peek. Phương thức element sẽ ném ra ngoại lệ nếu Queue là rỗng, còn peek sẽ trả về là null.



## **1.7Xóa phần tử đầu tiên từ Queue**

**Để xóa một phần tử chúng ta dùng phương thức remove**

****

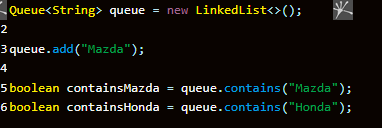
## **1.8 Xóa tất cả phần tử trong Queue**

Chúng ta sử dụng phương thức clear để xóa hết các phần tử trong Queue



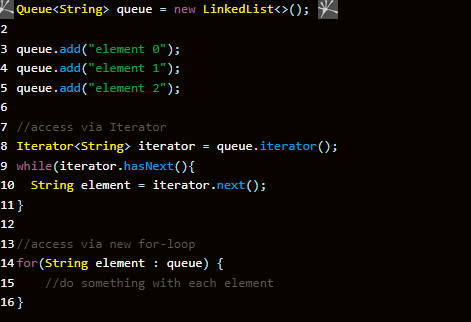
## **1.9 Kiểm tra phần tử đã có trong Queue**

Chúng ta sử dụng phương thức contains để kiểm tra phần tử đã tồn tại trong Queue chưa

****

## **2.0 Duyệt qua phần tử trong Queue**

Chúng ta có thể sử dụng Iterator hoặc foreach để duyệt qua các phần tử trong Queue



1. **What are different between Stack and Queue?**

Answer: : < CE161026> <Trần Tuấn Kiệt>

Stack và Queue đều là các cấu trúc dữ liệu không nguyên thủy (non-primitive). Sự khác biệt lớn nhất giữa Stack và Queue là Stack sử dụng phương thức LIFO (last in first out) để truy cập và thêm các phần tử dữ liệu trong khi Queue sử dụng phương thức FIFO (First in first out) để truy cập và thêm các phần tử dữ liệu.

Stack chỉ có một đầu mở để pushing và popping các phần tử dữ liệu, còn Queue có cả hai đầu mở để enqueuing và dequeuing các phần tử dữ liệu.

Stack và Queue là các cấu trúc dữ liệu được sử dụng để lưu trữ các yếu tố dữ liệu và nó dựa trên một số các ví dụ có thực trong cuộc sống hàng ngày của chúng ta. Ví dụ, Stack là một chồng đĩa CD, nơi bạn có thể lấy ra và đưa vào đĩa CD thông qua đỉnh của ngăn xếp đĩa CD. Tương tự, Queue là hàng đợi cho các vé của Nhà hát nơi người đứng ở vị trí đầu tiên, nghĩa là, phía trước hàng đợi sẽ được phục vụ trước và người mới đến sẽ xuất hiện ở phía sau hàng đợi.

1. **What is TreeSet? How to manage data using TreeSet? What is TreeSet used for?**

Answer: TreeSet là một trong những triển khai quan trọng nhất của giao diện SortedSet trong Java sử dụng Tree để lưu trữ. Thứ tự của các phần tử được duy trì bởi một tập hợp sử dụng thứ tự tự nhiên của chúng cho dù có cung cấp bộ so sánh rõ ràng hay không. Điều này phải phù hợp với bằng nếu nó được triển khai chính xác giao diện Đặt.

Manage date:

Hàm tạo này được sử dụng để xây dựng một đối tượng TreeSet trống trong đó các phần tử sẽ được lưu trữ theo thứ tự sắp xếp tự nhiên mặc định:

Set<String> ts = **new** TreeSet<>();

Hàm tạo này được sử dụng để xây dựng một đối tượng TreeSet chứa tất cả các phần tử từ tập hợp đã cho, trong đó các phần tử sẽ được lưu trữ theo thứ tự sắp xếp tự nhiên mặc định. Nói tóm lại, hàm tạo này được sử dụng khi cần chuyển đổi bất kỳ đối tượng Collection nào sang đối tượng TreeSet:

TreeSet ts = new TreeSet (Comparator comp);

Hàm tạo này được sử dụng để xây dựng một đối tượng TreeSet chứa tất cả các phần tử từ bộ sắp xếp đã cho, trong đó các phần tử sẽ được lưu trữ theo thứ tự sắp xếp tự nhiên mặc định. Tóm lại, hàm tạo này được sử dụng để chuyển đổi đối tượng SortedSet thành đối tượng TreeSet:

TreeSet t = new TreeSet (SortedSet s);

đây được coi là một trong những cấu trúc dữ liệu hiệu quả nhất để lưu trữ lượng dữ liệu khổng lồ đã được sắp xếp và thực hiện các thao tác trên đó. Tuy nhiên, các hoạt động như in N phần tử theo thứ tự đã sắp xếp mất O (N) thời gian.

1. **What is HashSet? How to manage data using HashSet? What is HashSet used for?**

Answer:

Lớp HashSet triển khai giao diện Set, được hỗ trợ bởi một bảng băm thực sự là một cá thể HashMap. Không có đảm bảo nào được thực hiện về thứ tự lặp lại của tập hợp có nghĩa là lớp không đảm bảo thứ tự không đổi của các phần tử theo thời gian. Lớp này cho phép phần tử null. Lớp cũng cung cấp hiệu suất thời gian không đổi cho các hoạt động cơ bản như thêm, xóa, chứa và kích thước giả sử hàm băm phân tán các phần tử đúng cách giữa các nhóm, chúng ta sẽ xem thêm trong bài viết.

Manage data:

Bạn xóa các phần tử khỏi HashSet bằng cách gọi phương thức remove(Object o);

Vd: // xóa phần tử PHP khỏi setA

        setA.remove("PHP");

Bạn có thể xóa tất cả các phần tử khỏi HashSet bằng phương thức clear().

[?](https://viettuts.vn/java-collection/hashset-trong-java)

|  |  |
| --- | --- |
| 1 | set.clear(); |

**Hashset là** một lớp giúp tạo ra một bộ sưu tập bằng cách **sử dụng** bảng băm **để** lưu trữ. **Hashset** mở rộng Tóm tắt; Tóm tắt thực hiện Set và Set mở rộng Bộ sưu tập. Nó có thể chứa các giá trị null và nó không **được** đồng bộ hóa. Hơn nữa, **Hashset** không duy trì thứ tự chèn và nó chỉ chứa các phần tử duy nhất.

1. **What is HashMap? How to manage data using HashMap? What is HashMap used for?**

Answer:

* Trước khi nói đến **HashMap**, chúng ta cần phải hiểu sơ lược về sự hình thành của nó, hay nói cách khác là sự bắt nguồn của HashMap. Nó xuất phát từ một **Map interface (java.util.Map)**, interface này biểu diễn sự ánh xạ giữa một trường gọi là "khóa" và một trường gọi là "giá trị" **Hashmap:** Cho phép chứa giá trị null, không hỗ trợ đồng bộ, với đặc điểm đặc trưng là thứ tự lặp (iteration order) không theo một quy tắc nào **(random)**.

Manage date:

**Phương thức: replace(key, newValue);**

Dùng để thay đổi giá trị của một khóa **K** bất kỳ đã tồn tại bên trong **HashMap**

**Phương thức: remove(key);**

Dùng để xóa một **Entry** ra khỏi **HashMap** dựa vào khóa **K**

* **HashMap**dùng làm từ điển còn được ứng dụng nhiều trong các chương trình và đặc biệt là ứng dụng trong quản lý các đối tượng mà đối tượng đó sử dụng một khóa (nhằm mục đích định danh) và trả về một thông tin về đối tượng đó

1. **What are different between TreeSet and HashSet and HashMap?**

Answer:

|  |  |  |
| --- | --- | --- |
| HashSet | LinkedHashSet | TreeSet |
| Cách thức làm việc? | HashSet sử dụng HashMap nội bộ để lưu trữ các phần tử. | LinkedHashSet sử dụng LinkedHashMap nội bộ để lưu trữ các phần tử. | TreeSet sử dụng TreeMap nội bộ để lưu trữ các phần tử. |
| Thứ tự của các phần tử (Order Of Elements) | HashSet không duy trì bất kỳ thứ tự các phần tử được thêm vào. | LinkedHashSet duy trì thứ tự chèn của các phần tử. Các phần tử được lưu trữ đúng như thứ tự chúng được chèn vào. | TreeSet duy trì thứ tự các phần tử theo bộ so sánh được cung cấp (Comparator). Nếu không có bộ so sánh được cung cấp, các phần tử sẽ được đặt theo thứ tự tăng dần tự nhiên của chúng. |
| Hiệu suất (Performance) | HashSet cho hiệu suất tốt hơn so với LinkedHashSet và TreeSet. | Hiệu suất của LinkedHashSet nằm giữa HashSet và TreeSet. Hiệu suất của nó hầu như tương tự như HashSet. Nhưng hơi chậm hơn vì nó cũng duy trì LinkedList nội bộ để duy trì trình tự chèn các phần tử. | TreeSet cho hiệu suất thấp hơn HashSet và LinkedHashSet vì nó phải sắp xếp các phần tử sau mỗi lần chèn và loại bỏ. |
| Thao tác thêm (Insertion), xóa (Removal) và truy xuất (Retrieval) phần tử | HashSet cho hiệu suất của lệnh O(1) để chèn, loại bỏ và truy xuất phần tử. | LinkedHashSet cũng cho hiệu suất của lệnh O(1) để chèn, loại bỏ và truy xuất phần tử. | TreeSet cho hiệu suất của lệnh O(log (n)) cho các thao tác chèn, loại bỏ và truy xuất phần tử. |
| So sánh các phần tử | HashSet sử dụng các phương thức equals() và hashCode() để so sánh các phần tử và do đó loại bỏ các phần tử có thể trùng lặp. | LinkedHashSet cũng sử dụng phương thức equals() và hashCode() để so sánh các phần tử. | TreeSet sử dụng phương pháp compare() hoặc compareTo() để so sánh các phần tử và do đó loại bỏ các phần tử có thể trùng lặp. Nó không sử dụng các phương thức equals () và hashCode () để so sánh các phần tử. |
| Phần tử Null | HashSet cho phép tối đa một phần tử null. | LinkedHashSet  cũng cho phép tối đa một phần tử null. | TreeSet không cho phép chứa phần tử null. Nếu bạn cố gắng để chèn null thành phần TreeSet, nó ném NullPointerException. |
| Sử dụng bộ nhớ | HashSet đòi hỏi ít bộ nhớ hơn LinkedHashSet và TreeSet vì nó chỉ sử dụng HashMap nội bộ để lưu trữ các phần tử của nó. | LinkedHashSet yêu cầu bộ nhớ nhiều hơn HashSet vì nó cũng duy trì LinkedList cùng với HashMap để lưu trữ các phần tử của nó. | TreeSet cũng yêu cầu bộ nhớ nhiều hơn HashSet vì nó cũng duy trì bộ so sánh để sắp xếp các phần tử cùng với TreeMap. |
| Khi nào sử dụng? | Sử dụng HashSet nếu bạn muốn danh sách không chứa phần tử trùng và không cần duy trì bất kỳ thứ tự các phần tử được chèn vào. | Sử dụng LinkedHashSet nếu bạn muốn danh sách không chứa phần tử trùng và muốn duy trì thứ tự chèn của các phần tử. | Sử dụng TreeSet nếu bạn muốn danh sách không chứa phần tử trùng và muốn sắp xếp các phần tử theo một số so sánh. |