TRƯỜNG ĐẠI HỌC CÔNG NGHIỆP HÀ NỘI

ﻣKHOA CÔNG NGHỆ THÔNG TINﻣ



**BÀI TẬP LỚN**

**LẬP TRÌNH JAVA**

**NGHIÊN CỨU CÁC KĨ THUẬT XỬ LÝ CỦA**

**Abstract Queue, ArrayBlockingQueue, ConcurrentLinkedQueue**

**Giáo viên hướng dẫn: Ths.Hoàng Quang Huy**

**Lớp: 202020503175008 (0503175.8) - K13**

**Nhóm: 16**

**Thành viên:**

1. **Phạm Văn Long**
2. **Bùi Văn Hà**
3. **Lưu Bá Minh**

**Hà Nội – 2021**

# LỜI NÓI ĐẦU

Hiện nay, ngành CNTT được xem là ngành đào tạo mũi nhọn hướng đến sự phát triển của công nghệ và khoa học kỹ thuật trong thời đại số hóa ngày nay. Công nghệ thông tin là một ngành học được đào tạo để sử dụng máy tính và các phần mềm máy tính để phân phối và xử lý các dữ liệu thông tin, đồng thời dùng để trao đổi, lưu trữ và chuyển đổi các dữ liệu thông tin dưới nhiều hình thức khác nhau. Việc sử dụng các ngôn ngữ lập trình như C, C++, PHP…. khá phát triển và được nâng cao hơn.

Đặc biệt Java là ngôn ngữ đã trở thành một trong những ngôn ngữ lập trình phát triển nhất trên thế giới. Bởi, bạn có thể tìm thấy Java tại các ứng dụng và môi trường của website trên tất cả các hệ điều hành và nền tảng khác nhau. Là một ngôn ngữ lập trình hướng đối tượng nên Java có nhiểu đặc trưng phù hợp với tất cả các **thiết kế website.**  Đặc điểm nổi bật nhất của Java là nó không phụ thuộc vào hệ điều hành và bộ xử lý. Điều này cho phép một ứng dụng viết bằng Java có thể được thực hiện trên bất kỳ hệ điều hành và hệ xử lý nào có hỗ trợ Java. Do vậy, người lập trình chỉ cần viết ứng dụng bằng Java đúng một lần, sau đó có thể sử dụng ứng dụng này trên các hệ điều hành khác nhau như Windows10, WindowsNT, Mac OS, Unix, … theo phương châm: “viết một lần, sử dụng ở bất kỳ đâu”.

Song song với đó, ta cần nghiên cứu sâu hơn về các kỹ thuật làm việc với Pair class trong JavaTuples. Bên dưới là nghiên cứu đề tài của nhóm 16, do trình độ chuyên môn, kinh nghiệm và kiến ​​thức của chúng em còn hạn chế, vì vậy chúng em rất mong nhận được lời khuyên của giáo viên và các bạn cho đề tài nghiên cứu của chúng em ngày càng hoàn thiện.

Chúng em xin chân thành cảm ơn!

LỜI NÓI ĐẦU 1

Chương 1: Tổng quát 6

1.1 Tổng quan về hàng đợi (Queue) 6

1.2 Khái quát về AbstractQueue, ArrayBlockingQueue và Concurrent*-*LinkedQueue 7

Chương 2: AbstractQueue 9

2.1 Khái niệm về AbstractQueue 9

2.2 AbstractQueue trong java 9

2.2.1 Cách khai báo trong AbstractQueue 9

2.2.2 Hệ thống phân cấp lớp 9

2.2.3 Cách tạo một AbstractQueue 10

2.2.4 Thêm một phần tử 10

2.2.5 Xóa một phần tử 11

2.2.6 Truy cập một phần tử 12

2.3 Các phương thức của AbstractQueue 12

2.3.1 Phương thức add(E e) 12

2.3.1 Phương thức addAll​(Collection<? extends E> c) 13

2.3.2 Phương thức thêm từ danh sách ArrayList 13

2.3.3 Phương thức thêm từ AbstractQueue 14

2.3.4 Phương thức clear() 14

2.3.5 Phương thức element() 14

2.3.6 Phương thức remove() 15

2.4 Kịch bản chạy và kết quả 16

2.4.1 Kịch bản chạy cho phương thức khởi tạo với constructor 16

2.4.2 Kịch bản chạy phương thức add(E e) 16

2.4.3 Kịch bản chạy phương thức addAll() 17

2.4.4 Kịch bản chạy phương thức element() 17

2.4.5 Kịch bản chạy phương thức remove() 18

Chương 3: ArrayBlockingQueue 19

3.1 Tổng quan về ArrayBlockingQueue 19

3.2 Hàm khởi tạo của ArrayBlockingQueue 20

3.3 Hàm khởi tạo của ArrayBlockingQueue 20

3.3.1 ArrayBlockingQueue(int capacity) 20

3.3.2 ArrayBlockingQueue(int capacity, boolean fair) 20

3.3.3 ArrayBlockingQueue(int capacity, boolean fair, Collection c) 21

3.4 Phương thức cơ bản 21

3.4.1 Thêm dữ liệu 21

3.4.2 Loại bỏ dữ liệu 22

3.4.3 Truy cập phần tử 23

3.4.4 Duyệt trình tự 24

3.5 Các phương thức của ArrayBlockingQueue 25

3.5.1 Phương thức *add(E e)* 25

3.5.2 Phương thức addAll(Colllection<E> c) 25

3.5.3 Phương thức *offer(E e)* 26

3.5.4 Phương thức offer(E e, long timeout, TimeUnit unit) 26

3.5.5 Phương thức *put(E e)* 27

3.5.6 Phương thức remove(), removeAll(), clear() 28

3.5.7 Phương thức contains(E e), containsAll(Collection c) 31

3.5.8 Phương thức drainTo(Collection c), drainTo(Collection c, int maxE) 32

3.5.9 Phương thức toArray(), toArray(T[] arr ) 33

3.5.10 Phương thức *remainingEmpty()* 35

3.5.11 Phương thức spliterator() 35

3.5.12 Phương thức *forEach()* 36

3.6 Kịch bản chạy chương trình 36

3.6.1 Kịch bản khởi tạo đối tượng 36

3.6.2 Kịch bản thêm phần tử 37

3.6.3 Kịch bản loại bỏ phần tử 40

3.6.4 Kịch bản kiểm tra tồn phần tử 42

3.6.5 Kịch bản chuyển đổi danh sách 43

3.6.6 Kịch bản chuyển đổi mảng 45

3.6.7 Kịch bản chuyển đổi dạng spliterator 46

3.6.8 Kịch bản lấy kích thước chưa sử dụng 47

Chương 4: ConcurrentLinkedQueue 48

4.1 Khái niệm chung về ConcurrentLinkedQueue 48

4.2 ConcurrentLinkedQueue trong java 48

4.2.1 Cách khai báo trong ConcurrentLinkedQueue 48

4.2.2 Hệ thống phân cấp lớp 48

4.2.3 Cách tạo một ConcurrentLinkedQueue 49

4.2.4 Thêm phần tử 49

4.2.5 Lấy giá trị 49

4.3 Các phương thức của ConcurrentLinkedQueue 51

4.3.1 Phương thức add​(E e) . 51

4.3.2 Phương thức addAll​(Collection<? extends E> c) . 51

4.3.3 Phương thức thêm từ danh sách ArrayList. 52

4.3.4 Phương thức thêm từ ConcurrentLinkedQueue. 52

4.3.6 Phương thức forEach​(Consumer<? super E> action). 53

4.3.7 Phương thức isEmpty() . 54

4.3.8 Phương thức iterator(). 54

4.3.9 Phương thức offer​(E e) . 55

4.3.10 Phương thức remove​(Object o) . 55

4.3.11 Phương thức removeAll​(Collection<?> c) . 56

4.3.12 Phương thức removeIf​(Predicate<? super E> filter) . 57

4.3.13 Phương thức spliterator() . 57

4.3.14 Phương thức toArray() . 58

4.3.15 Phương thức peek() 58

4.3.16 Phương thức element() 59

4.3.17 Phương thức poll(). 59

4.4 Kịch bản chạy và kết quả 60

4.4.1 Kịch bản chạy cho phương thức khởi tạo với constructor 60

4.4.2 Kịch bản chạy cho phương thức add() 60

4.4.3 Kịch bản chạy cho phương thức addAll() 61

4.4.4 Kịch bản chạy cho phương thức contains​​() 61

4.4.5 Kịch bản chạy cho phương thức forEach() 62

4.4.6 Kịch bản chạy cho phương thức isEmpty() 62

4.4.7 Kịch bản chạy cho phương thức iterator() 63

4.4.8 Kịch bản chạy cho phương thức offer() 63

4.4.9 Kịch bản chạy cho phương thức remove() 64

4.4.10 Kịch bản chạy cho phương thức removeAll() 64

4.4.11 Kịch bản chạy cho phương thức removeIf(). 65

4.4.12 Kịch bản chạy cho phương thức spliterator() 65

4.4.13 Kịch bản chạy cho phương thức toArray() 66

4.4.14 Kịch bản chạy cho phương thức lấy phần tử trong ConcurrentLinkedQueue 66

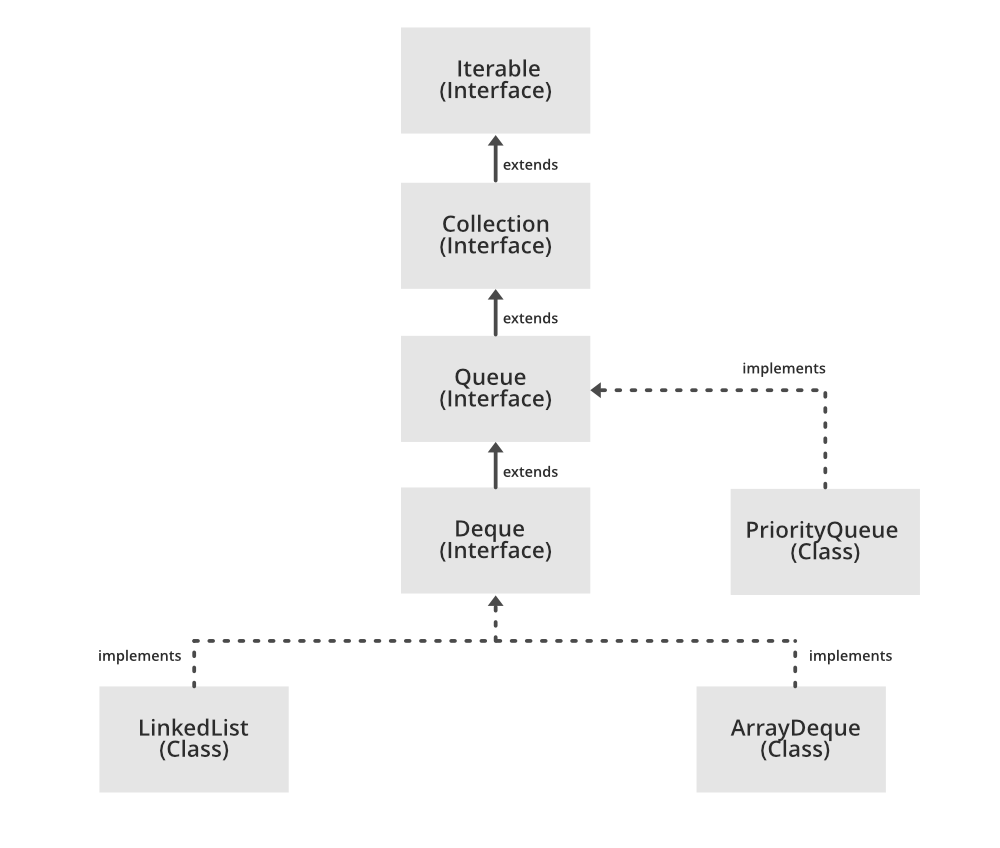
# Tổng quát

## Tổng quan về hàng đợi (Queue)

Queue (hàng đợi) là một **Interface** con của **Collection**, nó có đầy đủ các tính năng của **Collection**, nó khá giống với List, tuy nhiên mục đích sử dụng hơi khác nhau. Queue hoạt động theo cách thức **FIFO** (*First In First Out*). Trong FIFO, bạn chỉ có thể truy cập phần tử ở đầu hàng đợi, và khi loại bỏ phần tử nó loại phần tử đứng đầu hàng đợi. Nó giống như hàng người xếp hàng ở siêu thị, chỉ người đứng đầu hàng đợi mới được phục vụ, người mới đến sẽ được trèn vào hàng đợi, vị trí được trèn vào có thể không phải là cuối hàng. Vị trí phần từ được trèn vào phụ thuộc vào loại hàng đợi và độ ưu tiên của phần tử.

Đặc điểm:

* Là tập hợp cho phép các phần tử trùng lặp.
* Không cho phép phần tử null.

*Hình 1.1: Cách thức hoạt động của Queue (hàng đợi)*

*Hình 1.2: Mô hình phần cấp lớp hệ thống của Queue (hàng đợi)*

## Khái quát về AbstractQueue, ArrayBlockingQueue và Concurrent*-*LinkedQueue

- **AbstractQueue** lớp này cung cấp các triển khai khung của một số hoạt động hàng đợi(**Queue**). Các triển khai trong lớp này thích hợp khi triển khai cơ sở không cho phép các phần tử rỗng. Các phương thức *add*, *remove* và *element* tương ứng dựa trên *offer*, *poll* và *peek*, nhưng đưa ra các ngoại lệ thay vì chỉ ra lỗi thông qua trả về false hoặc null. Triển khai hàng đợi mở rộng lớp này phải xác định tối thiểu một phương thức Queue.offer (E) không cho phép chèn các phần tử null, cùng với các phương thức Queue.peek (), Queue.poll (), Collection.size (), Collection .iterator(). Thông thường, các phương thức bổ sung cũng sẽ bị ghi đè. Nếu không thể đáp ứng những yêu cầu này, hãy xem xét phân lớp con Abstract-Collection.

- ArrayBlockingQueue *extends* AbstractQueue và *implements* BlockingQueue. Hàng đợi chặn có giới hạn (ArrayBlockingQueue) được hỗ trợ bởi một mảng. Hàng đợi này sắp xếp thứ tự các phần tử FIFO (nhập trước xuất trước). Phần đầu của hàng đợi là phần tử đã ở trong hàng đợi lâu nhất. Phần cuối của hàng đợi là phần tử đã ở trên hàng đợi trong thời gian ngắn nhất. Các phần tử mới được chèn vào phần cuối của hàng đợi và các hoạt động truy xuất hàng đợi thu được các phần tử ở phần đầu của hàng đợi. Đây là một classic “*bounded buffer*”, trong đó một mảng có kích thước cố định chứa các phần tử do producers chèn vào và được customers trích xuất. Sau khi tạo, không thể thay đổi dung lượng. Nỗ lực đưa một phần tử vào một hàng đợi đầy đủ sẽ dẫn đến việc chặn hoạt động; các nỗ lực lấy một phần tử từ một hàng đợi trống cũng sẽ chặn tương tự. Lớp này hỗ trợ chính sách công bằng tùy chọn cho việc đặt hàng producers và customers đang chờ đợi. Theo mặc định, thứ tự này không được đảm bảo. Tuy nhiên, một hàng đợi được xây dựng với sự công bằng được đặt thành true sẽ cho phép các luồng truy cập theo thứ tự **FIFO**. Tính công bằng nói chung làm giảm thông lượng nhưng làm giảm sự thay đổi và tránh chết do thiếu phần tử. Lớp này và trình vòng lặp của nó thực hiện tất cả các phương thức tùy chọn của Collection interfaces và Iterator interfaces.

- ConcurrentLinkedQueue *extends* AbstractQueue và *implement* Queue.Một hàng đợi an toàn theo luồng không bị ràng buộc dựa trên các nút được liên kết. Hàng đợi này sắp xếp thứ tự các phần tử FIFO (nhập trước xuất trước). Phần đầu của hàng đợi là phần tử đã ở trong hàng đợi lâu nhất. Phần cuối của hàng đợi là phần tử đã ở trên hàng đợi trong thời gian ngắn nhất. Các phần tử mới được chèn vào phần cuối của hàng đợi và các hoạt động truy xuất hàng đợi thu được các phần tử ở phần đầu của hàng đợi. ConcurrentLinkedQueue là một lựa chọn thích hợp khi nhiều luồng sẽ chia sẻ quyền truy cập vào một tập hợp chung. Giống như hầu hết các triển khai tập hợp đồng thời khác, lớp này không cho phép sử dụng các phần tử null. Việc triển khai này sử dụng một thuật toán “wait-free” hiệu quả dựa trên một thuật toán được mô tả trong *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms của Maged M. Michael và Michael L. Scott*. Các trình lặp có tính nhất quán yếu, trả về các phần tử phản ánh trạng thái của hàng đợi tại một số thời điểm tại hoặc kể từ khi tạo trình lặp. Họ không ném *ConcurrentModification-Exception* và có thể tiến hành đồng thời với các hoạt động khác. Các phần tử có trong hàng đợi kể từ khi tạo trình lặp sẽ được trả về đúng một lần. Hãy lưu ý rằng, không giống như trong hầu hết các bộ sưu tập, phương thức kích thước KHÔNG phải là một hoạt động thời gian không đổi. Do tính chất không đồng bộ của các hàng đợi này, việc xác định số lượng phần tử hiện tại yêu cầu chuyển tải các phần tử và do đó có thể báo cáo kết quả không chính xác nếu tập hợp này được sửa đổi trong quá trình truyền tải. Ngoài ra, các hoạt động hàng loạt *addAll, removeAll, containsAll, retainAll, equals và toArray* không được đảm bảo thực hiện nguyên tử. Ví dụ: một trình lặp hoạt động đồng thời với một hoạt động addAll có thể chỉ xem một số phần tử được thêm vào. Lớp này và trình lặp của nó thực thi tất cả các phương thức tùy chọn của giao diện Queue và Iterator. Hiệu ứng nhất quán bộ nhớ: Như với các tập hợp đồng thời khác, các hành động trong một luồng trước khi đặt một đối tượng vào một ConcurrentLinkedQueue xảy ra trước các hành động tiếp theo khi truy cập hoặc xóa phần tử đó khỏi ConcurrentLinkedQueue trong một luồng khác.

# AbstractQueue

## Khái niệm về AbstractQueue

Lớp **AbstractQueue** trong Java là một phần của **Java Collection Framework** và implements **Collection** interface và lớp **AbstractCollection**.

Nó cung cấp các triển khai cơ bản của một số **Queue** operations.

Các triển khai trong lớp này thích hợp khi triển khai cơ sở không cho phép các phần tử rỗng.

Các phương thức add, remove và element dựa trên offer, poll và peek, respectively, nhưng đưa ra throw exeptions thay vì chỉ ra lỗi thông qua trả về false.

## AbstractQueue trong java

### Cách khai báo trong AbstractQueue

**public** **abstract** **class** AbstractQueue<E> **extends** AbstractCollection<E> **implements** Queue<E> {

//methods ...

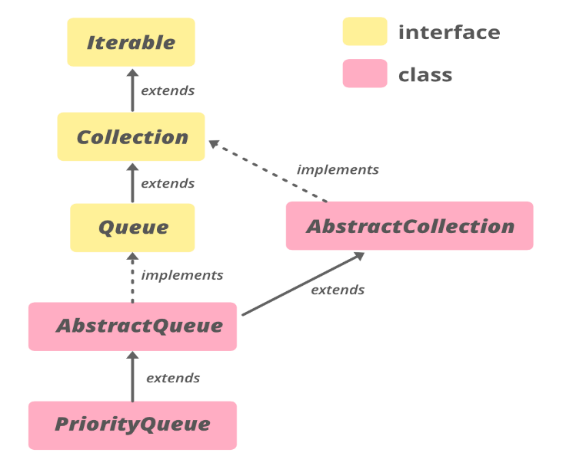
}

*Hình 2.1: Khai báo*

### Hệ thống phân cấp lớp

java.lang.Object

↳ java.util.AbstractCollection<E>

 ↳ Class AbstractQueue<E>

*Hình 2.2: Mô hình phân cấp lớp hệ thống của AbstractQueue*

Lớp này implements Iterable<E>, Collection<E>, Queue<E> interfaces và

extends AbstractCollection.

### Cách tạo một AbstractQueue

* Vì AbstractQueue là một lớp trừu tượng, vì vậy việc định nghĩa nó được triển khai ở các sub-class của nó. Dưới đây là hiển thị danh sách các lớp con có thể cung cấp việc triển khai. Để tạo ra nó, chúng ta cần import thư viện **java.uti.AbstractQueue**.
* Protected AbstractQueue(): hàm tạo mặc định, nhưng là trừu tượng, nó không cho phép tạo đối tượng AbstractQueue. Việc triển khai phải được cung cấp bởi một trong các lớp con của nó như là: [*ArrayBlockingQueue*](https://www.geeksforgeeks.org/arrayblockingqueue-class-in-java/#:~:text=ArrayBlockingQueue%20class%20is%20a%20bounded,result%20in%20the%20operation%20blocking.)*,*[*ConcurrentLinkedQueue*](https://www.geeksforgeeks.org/concurrentlinkedqueue-in-java-with-examples/)*,*[*DelayQueue*](https://www.geeksforgeeks.org/delayqueue-class-in-java-with-example/#:~:text=DelayQueue%20is%20a%20specialized%20Priority,queue%20whose%20time%20has%20expired.)*,*[*LinkedBlockingDeque*](https://www.geeksforgeeks.org/linkedblockingdeque-in-java-with-examples/)*,*[*LinkedBlockingQueue*](https://www.geeksforgeeks.org/linkedblockingqueue-class-in-java/#:~:text=LinkedBlockingQueue%20is%20an%20optionally%2Dbounded,the%20LinkedBlockingQueue%20will%20be%20unbounded.&text=It%20means%20that%20the%20head,elements%20present%20in%20this%20queue.)*,*[*LinkedTransferQueue*](https://www.geeksforgeeks.org/linkedtransferqueue-in-java-with-examples/#:~:text=The%20LinkedTransferQueue%20class%20in%20Java,functionality%20based%20on%20linked%20nodes.)*,*[*PriorityBlockingQueue*](https://www.geeksforgeeks.org/priorityblockingqueue-class-in-java/#:~:text=PriorityBlockingQueue%20is%20an%20unbounded%20blocking,resource%20exhaustion%20resulting%20in%20OutOfMemoryError.)*,*[*PriorityQueue*](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/), ***SynchronousQueue***.

AbstractQueue<E> obj = new ArrayBlockingQueue<E>();

### Thêm một phần tử

* Để thêm các phần tử AbstractQueue, nó cung cấp 2 phương thức:
  + Phương thức add(E e) sẽ chèn phần tử được chỉ định vào hàng đợi này nếu có thể thực hiện điều đó ngay lập tức mà không vi phạm các hạn chế về dung lượng. Nó trả về true khi thành công và bật ra IllegalStateException nếu hiện không có dung lượng.
  + Phương thức addAll(E e) thêm tất cả các phần tử trong specified collection vào queue.

// Since AbstractQueue is an abstract class

// create object using LinkedBlockingQueue

AbstractQueue<E> AQ1 = **new**

LinkedBlockingQueue<E>();

// Populating AQ

AQ1.add(E e);

AQ1.add(E e1);

// Since AbstractQueue is an abstract class

// create object using LinkedBlockingQueue

AbstractQueue<E> AQ2 = **new**

LinkedBlockingQueue<E>();

// adds elements of AQ1 in AQ2

AQ2.addAll(AQ1);

*Hình 2.3: Thêm phần tử*

### Xóa một phần tử

* Để loại bỏ các phần tử khỏi AbstractQueue, nó cung cấp các phương thức remove() và clear():
* Phương thức remove() trả về và loại bỏ phần tử đầu của queue này. Nếu chúng ta truyền vào giá trị cần remove vào phương thức thì phương thức sẽ check trong queue có tồn tại nó không và nếu có thì sẽ thực hiện xóa ngược lại thì queue sẽ vẫn giữ nguyên.
* Phương thức clear() xóa tất cả các phần tử khỏi queue này. Queue sẽ trống sau khi gọi hàm này.

// remove head element

AQ1.remove();

// remove element

AQ1.remove(E e);

// remove all elements

AQ2.clear();

*Hình 2.4: Xóa phần tử*

### Truy cập một phần tử

* Phương thức element() của AbstractQueue truy xuất nhưng không loại bỏ, phần tử đầu tiên của queue.

AbstractQueue<E> AQ1 = **new** LinkedBlockingQueue<E>();

AQ1.element();

*Hình 2.5: Truy cập một phần tử*

## Các phương thức của AbstractQueue

### Phương thức add(E e)

* Phương thức add(E e) của AbstractQueue sẽ chèn thêm phần tử được chỉ định vào queue này nếu có thể làm như vậy ngay lập tức mà không vi phạm giới hạn dung lượng. Nó trả về true khi thành công và bật ra IllegaStateException nếu không có dung lượng.

- Cú pháp: *public boolean add(E e)*

* Parameters: Phương thức chấp nhận một tham số bắt buộc e là phần tử được chèn vào queue.
* Returns: Phương thức này trả về true nếu phần tử được chèn vào queue ngược lại trả lại false.
* Exception: Phương thức này bật ra các Exception sau:

+ **IllegalStateException**: if the element cannot be added at this time due to capacity restrictions

+ **NullPointerException**: if the specified element is null

+ **ClassCastException** – if the class of the specified element prevents it from being added to this queue

+ **IllegalArgumentException** – if some property of this element prevents it from being added to this queue

*Hình 2.6: Phương thức add*

/\*\*

\* Phương thức <b>thêm</b> một số int n vào queue. <br>

\* Phương thức này sẽ chèn số n vào cuối queue. <br>

\* Phương thức này sẽ trả về true nếu thêm thành công

\* ngược lại trả về false

\* **@param** AQ

\* **@param** n

\* **@return**

\*/

**public** **static** **boolean** add(AbstractQueue<Integer> AQ, **int** n)

{

**return** AQ.add(n);

}

### Phương thức addAll​(Collection<? extends E> c)

Phương thức addAll () của AbstractQueue được sử dụng để chèn tất cả các phần tử của Collection, được truyền dưới dạng tham số cho phương thức này, vào cuối AbstractQueue. Việc chèn phần tử theo thứ tự như được trả về bởi collections iterator.

### Phương thức thêm từ danh sách ArrayList

/\*\*

\* - Phương thức <b>thêm</b> một danh sách số vào hàng đợi.</br>

\* - Phương thức này sẽ chèn một danh sách số vào cuối của hàng đợi.</br>

\* - Phương thức này sẽ trả về true nếu thêm thành công và nó

\* sẽ trả về false nếu thêm thất bại.

\* **@param** queue

\* **@param** listInteger

\* **@return**\*/

**public** **static** **boolean** addAll(AbstractQueue<Integer> queue,

ArrayList<Integer> listInteger) {

// Phương thức addAll() sẽ chèn tất cả các phần tử của

// danh sách vào cuối của hàng đợi.

**return** queue.addAll(listInteger);

}

*Hình 2.7: Phương thức thêm từ danh sách ArrayList*

### Phương thức thêm từ AbstractQueue

/\*\*

\* - Phương thức <b>thêm</b> một hàng đợi newQueue vào một

\* hàng đợi khác queue.</br>

\* - Phương thức này sẽ chèn một hàng đợi newQueue vào cuối của hàng đợi

\* queue.</br>

\* - Phương thức này sẽ trả về true nếu thêm thành công và nó sẽ trả về false nếu thêm thất bại.

\*

\* **@param** queue

\* **@param** newQueue

\* **@return**

\*/

**public** **static** **boolean** addAll(AbstractQueue<Integer> queue,

AbstractQueue<Integer> newQueue) {

// Phương thức addAll() sẽ chèn tất cả các phần tử của

// newQueue vào cuối của hàng đợi queue.

**return** queue.addAll(newQueue);

}

*Hình 2.8: Phương thức thêm từ AbstractQueue*

### Phương thức clear()

* Phương thức clear() của AbstractQueue loại bỏ tất cả các phần tử khỏi queue. Queue đó sẽ trống sau khi call returns.

/\*\*

\* - Phương thức xóa toàn bộ phần tử của AbstractQueue

\* **@param** AQ

\*/

**public** **static** **void** clear(AbstractQueue<Integer> AQ) {

// Phương thức clear() sẽ xóa tất cả các phần tử của AQ

AQ.clear();

}

*Hình 2.9: Phương thức clear()*

### Phương thức element()

* Phương thức element() của AbstractQueue truy xuất nhưng không loại bỏ phần tử đầu của Queue.

*Hình 2.10: Phương thức element()*

/\*\*

\* - Phương thức truy xuất phần tử đầu của AbstracQueue

\* **@param** AQ

\* **@return**

\*/

**public** **static** Integer element(AbstractQueue<Integer> AQ) {

**return** AQ.element();

}

### Phương thức remove()

* Phương thức remove() của AbstractQueue loại bỏ phần tử có trong queue.
* Nếu không truyền tham số mặc định nó sẽ loại bỏ đi phần tử đầu của queue.
* Nếu truyền tham số thì nó sẽ loại bỏ đi phần tử tương đương với phần tử truyền vào trong queue nhưng nếu có nhiều phần tử trùng thì nó sẽ remove đi phần tử trùng với tham số truyền vào đầu tiên.

/\*\*

\* - Phương thức xóa 1 phần tử trong AbstractQueue

\* **@param** AQ

\* **@param** n

\* **@return**

\*/

**public** **static** **boolean** remove(AbstractQueue<Integer> AQ, **int** n) {

**return** AQ.remove(n);

}

*Hình 2.11: Phương thức remove*

## Kịch bản chạy và kết quả

### Kịch bản chạy cho phương thức khởi tạo với constructor

* Kịch bản chạy:

**public** **static** **void** main(String[] argv) **throws** Exception {

// Since AbstractQueue is an abstract class

// create object using LinkedBlockingQueue

AbstractQueue<Integer> AQ1 = **new** LinkedBlockingQueue<Integer>();

// Populating AQ

AQ1.add(10);

AQ1.add(20);

AQ1.add(30);

AQ1.add(40);

AQ1.add(50);

// print AQ

System.***out***.println("AbstractQueue contains: " + AQ1);

}

*Hình 2.12: Kịch bản chạy cho phương thức khởi tạo với constructor*

* Kết quả:

AbstractQueue contains: [10, 20, 30, 40, 50]

*Hình 2.13: Kết quả*

### Kịch bản chạy phương thức add(E e)

* Kịch bản chạy:

**public** **static** **void** main(String[] argv) **throws** Exception {

// Since AbstractQueue is an abstract class

// create object using LinkedBlockingQueue

AbstractQueue<Integer> AQ1 = **new** LinkedBlockingQueue<Integer>();

System.***out***.println(AQ1.add(5));

System.***out***.println("AbstractQueue contains: " + AQ1);

}

*Hình 2.14: Kịch bản chạy phương thức add*

* Kết quả chạy:

true

AbstractQueue contains: [15]

*Hình 2.14: Kết quả*

### Kịch bản chạy phương thức addAll()

* Kịch bản chạy:

**public** **static** **void** main(String[] argv) **throws** Exception {

// Since AbstractQueue is an abstract class

// create object using LinkedBlockingQueue

AbstractQueue<Integer> AQ1 = **new** LinkedBlockingQueue<Integer>();

ArrayList<Integer> arrList = **new** ArrayList<Integer>();

arrList.add(1);

arrList.add(2);

System.***out***.println(*addAll*(AQ1 ,arrList));

// print AQ

System.***out***.println("AbstractQueue contains: " + AQ1);

}

*Hình 2.15: Kịch bản chạy phương thức addAll*

* Kết quả chạy:

true

AbstractQueue contains: [1, 2]

*Hình 2.16: Kết quả*

### Kịch bản chạy phương thức element()

* Kịch bản chạy:

**public** **static** **void** main(String[] argv) **throws** Exception

{

// Since AbstractQueue is an abstract class

// create object using LinkedBlockingQueue

AbstractQueue<Integer> AQ1 = **new** LinkedBlockingQueue<Integer>();

// Populating AQ

AQ1.add(10);

AQ1.add(20);

AQ1.add(30);

*Hình 2.17: Kịch bản chạy phương thức element*

AQ1.add(40);

AQ1.add(50);

System.***out***.println(*element*(AQ1));

System.***out***.println("AbstractQueue contains: " + AQ1;

}

* Kết quả:

10

AbstractQueue contains: [10, 20, 30, 40, 50]

*Hình 2.18: Kết quả*

### Kịch bản chạy phương thức remove()

* Kịch bản chạy:

**public** **static** **void** main(String[] argv) **throws** Exception {

// Since AbstractQueue is an abstract class

// create object using LinkedBlockingQueue

AbstractQueue<Integer> AQ1 = **new** LinkedBlockingQueue<Integer>();

// Populating AQ

AQ1.add(10);

AQ1.add(20);

AQ1.add(30);

AQ1.add(40);

AQ1.add(50);

System.***out***.println("AbstractQueue contains: " + AQ1);

System.***out***.println("Remove status: " + *remove*(AQ1, 20));

System.***out***.println("AbstractQueue contains: " + AQ1);

}

*Hình 2.19: Kịch bản chạy phương thức remove*

* Kết quả chạy:

AbstractQueue contains: [10, 20, 30, 40, 50]

Remove status: true

AbstractQueue contains: [10, 30, 40, 50]

*Hình 2.20: Kết quả*

# ArrayBlockingQueue

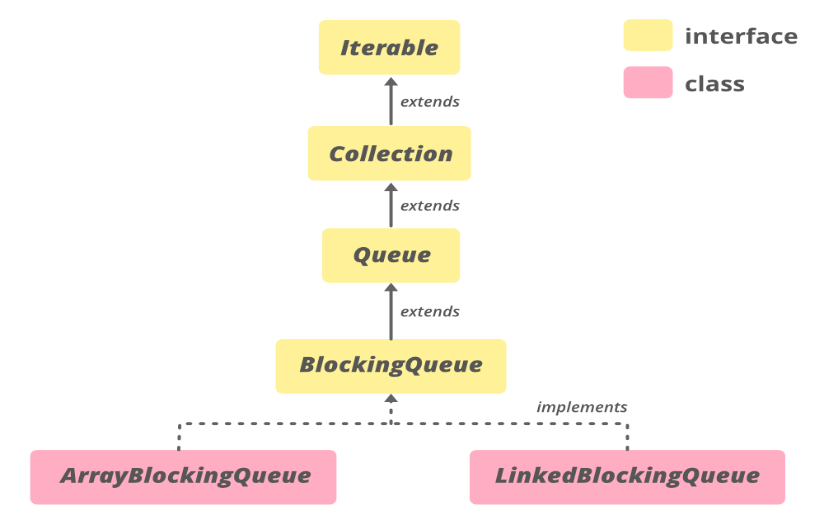
## Tổng quan về ArrayBlockingQueue

ArrayBlockingQueue là một hàng đợi có kích thước cố định dưới dạng mảng. Kích thước hàng đợi sẽ được cố định, đồng nghĩa là không thể thay đổi sau khi khởi tạo. Nếu cố thêm dữ liệu vào hàng đợi, điều đó sẽ ngăn cản sự hoạt động của nó. Và tương tự với việc lấy dữ liệu từ một hàng đợi rỗng thì cũng sẽ bị chặn.

Sự rằng buộc của ArrayBlockingQueue có thể bỏ qua lưu trữ ban đầu thông qua tham số được truyền tại constructor trong ArrayBlockingQueue. Hiểu đơn giản là yêu cầu đầu vào của ArrayBlockingQueue sẽ được truyền qua tham số. ArrayBlockingQueue có dạng FIFO (first – in – first - out). Dữ liệu được thêm vào sẽ được đẩy về cuối hàng đợi và hàng đợi sẽ thực thi các giá trị phía đầu hàng.

ArrayBlockingQueue được thực hiện tất cả các phương thức của Collection và Iterator interface. *Interface* là một cách khai báo phương thức, nhưng không chứa logic. Các class khác sẽ kế thừa interface và dùng các phương thức được interface khai báo sẵn. Interface có thể implement nhiều interface.

*Implement* là một các hướng đa kế thừa được java cung cấp. Một class chỉ có thể extends một class khác. Trong khi đó, một class có thể implement nhiều interface. Đồng nghĩa với việc, sẽ mở rộng dễ dàng hơn. Giảm tính phụ thuộc giữa các phương thức. Vì implement chỉ kế thừa phương thức, chứ không kế thừa logic, vì vậy implement có thể tái sử dụng nhiều lần mà không bị conflict code logic.

*Hình 3.1: Mô hình phân cấp lớp hệ thống của ArrayBlockingQueue*

## Hàm khởi tạo của ArrayBlockingQueue

**public class** ArrayBlockingQueue<E> **extends** AbstractQueue<E> **implements** BlockingQueue<E>, Serializable

*Hình 3.2: Khởi tạo ArrayBlockingQueue*

## Hàm khởi tạo của ArrayBlockingQueue

### **ArrayBlockingQueue(int capacity)**

* **Tạo một ArrayBlockingQueue với kích thước được cố địch từ ban đầu (capacity) và cách thức truy cập mặc định.**
* **Cú pháp:**

**// Khai báo đối tượng hàng đợi kiểu Interger**

**int capacity = 10;**

**ArrayBlockingQueue<Integer> adp = new ArrayBlockingQueue<>(capacity)**

***Hình 3.3: Khởi tạo một tham số***

### **ArrayBlockingQueue(int capacity, boolean fair)**

* **Tạo một ArrayBlockingQueue với kích thước được cố địch từ ban đầu và cách thức truy cập mặc định. Nếu giá trị của *fair* là true thì truy cập hàng đợi cho các chuỗi bị chặn khi chèn hoặc loại bỏ, được xử lý theo thứ tự FIFO (First In First Out). Nếu là *false* thì thứ tự truy cập là không xác định.**
* **Cú pháp:**

**// Khai báo đối tượng hàng đợi kiểu Interger**

**int capacity = 10;**

**Boolean fair = true;**

**ArrayBlockingQueue<Integer> adp = new ArrayBlockingQueue<>(capacity, fair)**

***Hình 3.4: Khởi tạo hai tham số***

### **ArrayBlockingQueue(int capacity, boolean fair, Collection c)**

* **Tạo một ArrayBlockingQueue với kích thước được cố địch từ ban đầu và cách thức truy cập mặc định chứa các phần tử được lấy từ biến *c*, được thêm vào theo thứ tự duyệt của trình lặp của tập hợp. Nếu giá trị của *fair* là true thì truy cập hàng đợi cho các chuỗi bị chặn khi chèn hoặc loại bỏ, được xử lý theo thứ tự FIFO (First In First Out). Nếu là *false* thì thứ tự truy cập là không xác định**
* **Cú pháp:**

**// Khai báo đối tượng hàng đợi kiểu Interger**

**int capacity = 10;**

**Boolean fair = true;**

**Collection c = new Collection();**

**ArrayBlockingQueue<Integer> adp = new ArrayBlockingQueue<>(capacity, fair, c)**

***Hình 3.5: Khởi tạo ba tham số***

## Phương thức cơ bản

### Thêm dữ liệu

* *add(E e)* là phương thức thêm phần tử được truyền qua tham số vào hàng đợi đang thực thi phương thức này. Nếu việc thêm phần tử mới vào hàng đợi vượt quá giới hạn lưu trữ hiện có của hàng đợi thì sẽ đưa ra lỗi **IllegalStateException**. Phương thức sẽ trả về **true** nếu thêm thành công phần tử, ngược lại sẽ đưa ra lỗi **IllegalStateException.**
* Cú pháp:

**public static void** main(String[] args)

{

   // Định nghĩa kích thước của hàng đợi

**int** capacity = 15;

   // Tạo đối tượng hàng đợi

ArrayBlockingQueue<Integer> abq = **new** ArrayBlockingQueue<Integer>(capacity);

   // Thêm giá trị

   abq.add(1);

   abq.add(2);

   abq.add(3);

   // Hiển thị hàng đợi

   System.**out**.println("ArrayBlockingQueue:" + abq);

 }

*Hình 3.6: Thêm dữ liệu*

* Kết quả

ArrayBlockingQueue:[1, 2, 3]

*Hình 3.7: Kết quả*

### Loại bỏ dữ liệu

* *remove(Objec o)* là phương thức loại bỏ một phần tử hoặc một thực thể của chính hàng đợi đang gọi tới nó. Nếu hàm trả về *True* đồng nghĩa với việc là một phần tử hoặc một thực thể đã bị loại bỏ khỏi danh sách, ngược lại trả về *False* là không có phần tử nào bị loại bỏ.
* Cú pháp:

**public static void** main(String[] args)

{

// Định nghĩa kích thước của hàng đợi

**int** capacity = 15;

   // Tạo đối tượng hàng đợi

ArrayBlockingQueue<Integer> abq = **new** ArrayBlockingQueue<Integer>(capacity);

// Thêm giá trị

abq.add(1);

abq.add(2);

    abq.add(3);

*Hình 3.8: Loại bỏ dữ liệu*

// Hiển thị hàng đợi

System.**out**.println("ArrayBlockingQueue:" + abq);

// Loại bỏ giá trị 2

**boolean** res = abq.remove(2);

// Hiển thị hàng đợi

System.**out**.println("Loại bỏ giá trị 2 :" + res);

// Hiển thị hàng đợi

System.**out**.println("ArrayBlockingQueue:" + abq);

   // Loại bỏ tất cả giá trị của hàng đợi

  abq.clear();

// Hiển thị lại hàng đợi

System.**out**.println("ArrayBlockingQueue:" + abq);

}

* Kết quả

ArrayBlockingQueue:[1, 2, 3]

Loại bỏ giá trị 2: true

ArrayBlockingQueue:[1, 3]

ArrayBlockingQueue:[]

*Hình 3.9: Kết quả*

### Truy cập phần tử

* *peek()* là phương thức được cung cấp bởi **Queue** interface, sử dụng để trả về giá trị đầu tiên của hàng đợi, nếu không có thì sẽ trả về giá trị *null.* Phương thức này chỉ truy cập, không làm loại bỏ phần tử.
* Cú pháp:

**public** **static** **void** main(String[] args)

{

// Định nghĩa kích thước của hàng đợi

**int** capacity = 5;

// Tạo đối tượng hàng đợi

ArrayBlockingQueue<Integer> queue = **new** ArrayBlockingQueue<Integer>(capacity);

// Thêm giá trị vào hàng đợi

queue.add(23);

queue.add(32);

queue.add(45);

queue.add(12);

*Hình 3.10: Truy cập phần tử*

// Hiển thị hàng đợi

System.**out**.println("ArrayBlockingQueue:" + queue);

// Hiển thị giá trị vị trí đầu của hàng đợi

// Bằng cách sử dụng peek()

System.**out**.println("Head of queue " + queue.peek());

}

* Kết quả

ArrayBlockingQueue:[23, 32, 45, 12]

Head of queue: 23

*Hình 3.11: Kết quả*

### Duyệt trình tự

* *iterator()* là phương thức của lớp **ArrayBlockingQueue**, được sử dụng để trả về một trình tự lắp của các phần tử giống như hàng đợi (theo một trình tự nhất định). Các phần tử được trả về từ phương thức này chứa các phần tử theo thứ tự từ đầu đến cuối của hàng đợi. Sẽ trả về thông báo lỗi, nếu hàng đợi rỗng.
* Cú pháp:

**public** **static** **void** main(String[] args)

{

// Định nghĩa kích thước của hàng đợi

**int** capacity = 5;

// Tạo đối tượng hàng đợi

ArrayBlockingQueue<String> queue = **new** ArrayBlockingQueue<String>(capacity);

// Thêm giá trị vào hàng đợi

queue.offer(“Nhan Vien”);

queue.offer(“Giam Doc”);

queue.offer(“Bao Ve”);

queue.offer(“Nhan Su”);

// Hiển thị hàng đợi

System.**out**.println("ArrayBlockingQueue:" + queue);

// Gọi hàm iterator() và tạo iterator

Iterator interatorValue = queue.iterator();

System.**out**.println("\nDanh sách giá trị:");

**while** (iteratorValues.hasNext()) {

System.**out**.println(iteratorValues.next());

    }

}

*Hình 3.12: Duyệt trình tử*

* Kết quả

ArrayBlockingQueue:[Nhan Vien, Giam Doc, Bao Ve, Nhan Su]

Danh sách giá trị:

Nhan Vien

Giam Doc

Bao Ve

Nhan Su

Hình 3.13: Kết quả

## Các phương thức của ArrayBlockingQueue

### Phương thức *add(E e)*

* Phương thức dùng để thêm phần tử vào cuối hàng đợi thông qua tham số truyền vào hàm. Nếu thêm thành công sẽ trả về *True,* ngược lại sẽ trả vè thông báo lỗi *IllegalStateException.*
* Cú pháp:

/\*\*

\* Phương thức sẽ thêm phần tử mới vào hàng đợi với vị trí là cuối cùng

\* Trả về True nếu thêm thành công, ngược lại sẽ trả về thông báo

\* lỗi<br>

\* Create by: BVHA (12/05/2021)

\* @param <E>

\* @param abq

\* @param e - Phần tử để thêm vào hàng đợi

\* **@return**\*/

**public** **static** <E> **boolean** addElement(ArrayBlockingQueue<E> abq, E e)

{

**return** abq.add(e);

}

*Hình 3.14: Phương thức add*

### Phương thức addAll(Colllection<E> c)

* Phương thức dùng để thêm phần tử từ danh sách được truyền qua tham số vào cuối hàng đợi. Nếu thêm thành công sẽ trả về *true,* ngược lại sẽ trả vè thông báo lỗi *IllegalStateException.*
* Cú pháp:

/\*\*

\* Phương thức thực dùng để thêm mới một danh sách phần tử vào hàng

\* đợi. Trả về <i>True</i> nếu thêm tất cả phần tử từ danh sách vào

\* hàng đợi thành công. Ngược lại, trả về thông báo lỗi. <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@param** collec - Danh sách thêm vào hàng đợi

\* **@return**

\*/

**public** **static** <E> **boolean** addAllElement(ArrayBlockingQueue<E> abq, Collection<E> collec)

{

**return** abq.addAll(collec);

}

*Hình 3.15: Phương thức addAll*

### Phương thức *offer(E e)*

* Phương thức thêm phần tử vào cuối hàng đợi được truyền qua tham số. Trả về *True* nếu thêm thành công, ngược lại trả về *False (hàng đợi đầy).* Phương thức này khác với phương thức *add,* bởi vì phương thức *add* sẽ thông báo lỗi khi hàng đợi đầy, còn phương thức *offer* sẽ trả về *false.*
* Cú pháp:

/\*\*

\* Phương thức được dùng để thêm mới phần tử <b>e</b> vào hàng đợi với

\* vị trí là cuối cùng Trả về <i>True</i> nếu thêm thành công

\* ngược lại sẽ trả về <i>False</i> <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq

\* **@param** e - Phần tử để thêm vào hàng đợi

\* **@return**

\*/

**public** **static** <E> **boolean** addElementByOffer(ArrayBlockingQueue<E>

abq, E e)

{

**return** abq.offer(e);

}

*Hình 3.16: Phương thức offer một tham số*

### Phương thức offer(E e, long timeout, TimeUnit unit)

* Phương thức thêm phần tử vào cuối hàng đợi được truyền qua tham số. Nếu hàng đợi đầy, phương thức sẽ đợi một khoảng thời gian cho đến khi có khoảng trống trong hàng đợi. Trả về *true* nếu thêm thành công, ngượi lại sẽ thực hiện chờ đợi khoảng trống trong hàng đợi. Phương thức rất phù hợp khi chúng ta muốn đợi hàng đợi loại bỏ phần tử của nó, khi hàng đợi không đầy sẽ thực hiện thêm phần tử vào hàng đợi .
* Cú pháp:

/\*\*

\* Phương thức được dùng để thêm mới phần tử <b>e</b> vào hàng đợi. Nếu

\* hàng đợi đầy,thì hàm sẽ thực hiện đợi một khoảng thời gian

\* <i>time</i>, với cách tính thời gian dựa vào <i>unit</i> Trả về

\* <i>True</i> nếu thêm thành công, ngược lại trả về <i>False</i> <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@param** e - Phần tử để thêm vào hàng đợi

\* **@param** time - Thời gian chờ

\* **@param** unit - Đơn vị tính thời gian chờ

\* **@return**

\*/

**public** **static** <E> **boolean** addElementByOffer(ArrayBlockingQueue<E> abq, E e, **long** time, TimeUnit unit)

{

**try** {

**return** abq.offer(e, time, unit);

} **catch** (InterruptedException e1) {

e1.printStackTrace();

**return** **false**;

}

}

*Hình 3.17: Phương thức offer ba tham số*

### Phương thức *put(E e)*

* Phương thức thêm phần tử vào cuối hàng đợi nếu hàng đợi không đầy, ngược lại sẽ đợi cho đến khi có thể thêm mới phần tử vào cuối hàng đợi.

*Hình 3.18: Phương thức put*

/\*\*

\* Phương thức dùng để thêm phần tử mới vào hàng đợi. Nếu hàng đợi

\* chưa đầy, thì thực hiện thêm mới phần tử vào cuối hàng đợi.

\* Ngược lại, sẽ đợi cho đến khi có vị trí thêm mới. <br>

\* **@param** abq

\* **@param** e

\* **@throws** InterruptedException

\*/

**public** **static** <E> **void** addElementByPut(ArrayBlockingQueue<E> abq, E e) **throws** InterruptedException

{

abq.put(e);

}

### Phương thức remove(), removeAll(), clear()

* *remove(E e)*

Phương thức thực hiện loại bỏ phần tử trong hàng đợi theo giá trị của biến truyền vào. Trả về *true* nếu loại bỏ thành công, ngược lại trả về *false.*

* Cú pháp:

/\*\*

\* Phương thức loại bỏ một phần tử theo giá trị <b>e</b> tồn tại trong

\* hàng đợi, thứ tự từ FIFO. Trả về <i>True</i>nếu loại bỏ thành công,

\* ngược lại trả về <i>False</i> <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợ

\* **@param** e - Phần tử loại bỏ khỏi hàng đợi

\* **@return**

\*/

**public** **static** <E> **boolean** removeElementByValue(ArrayBlockingQueue<E> abq, E e)

{

**return** abq.remove(e);

}

*Hình 3.19: Phương thức remove có tham số*

* *remove()*

Phương thức trả về phần tử đầu và đồng thời loại bỏ khỏi hàng đợi. Nếu hàng đợi rỗng, trả về NoSuchElementException.

* Cú pháp:

/\*\*

\* Phương thức trả về phần tử đầu tiên, đồng thời loại bỏ phần tử đó khỏi

\* hàng đợi. Trả về thông báo lỗi nếu hàng đợi rỗng.<br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@return**

\*/

**public** **static** <E> E removeElementFirst(ArrayBlockingQueue<E> abq)

{

**return** abq.remove();

}

*Hình 3.20: Phương thức remove không tham số*

* *removeAll(Collection c)*

Phương thức dùng để loại bỏ từ một tất cả phần tử của danh sách dựa vào phần tử của danh sách truyền qua tham số. Trả về *true* nếu hàm được thực thi, ngược lại trả về *NullPointerException* nếu danh sách truyền vào *null .*

* Cú pháp:

/\*\*

\* Phương thức loại bỏ các phần tử dựa vào danh sách phần tử được truyền

\* vào. Trả về <i>true</i> nếu hàm được thực thi, ngược lại trả về thông

\* báo lỗi nếu danh sách được truyền vào là <i>null</i><br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq

\* **@param** c

\* **@return**

\*/

**public** **static** <E> **boolean** removeAllElement(ArrayBlockingQueue<E> abq, Collection<E> c)

{

**return** abq.removeAll(c);

}

*Hình 3.21: Phương thức removeAll*

* *removeIf()*

Phương thức dùng để loại bỏ phần tử của danh sách dựa vào hàm lọc điều kiện, nó được truyền qua tham biến của hàm. Trả về *true* nếu có khả năng loại bỏ phần tử, ngược lại trả về *NullPointerException* nếu hàng đợi *null.*

* Cú pháp:

/\*\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@return**

\*/

**public** **static** <E> E removeElementFirst(ArrayBlockingQueue<E> abq)

{

**return** abq.removeIf(Predicate filter);

}

*Hình 3.22: Phương thức removeIf*

* *clear()*

Phương thức loại bỏ tất cả phần tử của hàng đợi. Sau khi thực thi, hàng đợi sẽ rỗng.

* Cú pháp:

/\*\*

\* Phương thức loại bỏ tất cả các phần tử của hàng đợi. Hàng đợi sẽ rỗng

\* sau khi gọi hàm này <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\*/

**public** **static** <E> **void** clear(ArrayBlockingQueue<E> abq) {

abq.clear();

}

*Hình 3.23: Phương thức clear*

### Phương thức contains(E e), containsAll(Collection c)

* *contains()*

Phương thức kiểm tra tồn tại của phần tử trong tham số của hàm có trong hàng đợi hay không. Nếu tồn tại trả về *true,* ngược lại trả về *false.*

* Cú pháp:

/\*\*

\* Phương thức kiểm tra tồn tại của phần tử <b>e</b> có trong hàng đợi.

\* Trả về True nếu tồn tại, ngược lại trả về False <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@param** e - Phần tử để kiểm tra tồn tại

\* **@return**

\*/

**public** **static** <E> **boolean** containElement(ArrayBlockingQueue<E> abq, E e) {

**return** abq.contains(e);

}

*Hình 3.23: Phương thức contains*

* *containsAll()*

Phương thức dùng để kiểm tra xem hai danh sách có cùng chứa chung những phần tử hay không. Hàm sẽ nhận một danh sách truyền qua tham số và trả về *true* nếu tất cả phần tử của danh sách hiện tại tồn tại danh sách được truyền vào kia, ngược lại trả về *false.*

* Cú pháp:

/\*\*

\* Phương thức kiểm tra tồn tại của tất cả phần tử trong <b>abq2</b> tồn

\* tại trong <b>abq</b> Trả về <i>True</i> nếu tồn tại, ngược lại trả về

\* <i>False</i> <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@param** abq2 - Danh sách giá trị để kiểm tra tồn tại trong hàng đợi

\* **@return** boolean

\*/

**public** **static** <E> **boolean** containAllElement(ArrayBlockingQueue<E> abq, Collection<E> abq2) {

**return** abq.containsAll(abq2);

}

*Hình 3.24: Phương thức containsAll*

### Phương thức drainTo(Collection c), drainTo(Collection c, int maxE)

* *drainTo(Collection c)*

Phương thức loại bỏ tất cả phần tử từ hàng đợi và thêm chúng vào danh sách. Trả về số lượng phần tử của drain. Và sẽ báo lỗi trong các trường hợp sau:

* ***UnsupportedOperationException***: Nếu không được hỗ trợ bởi danh sách.
* ***ClassCastException****:* Nếu lớp của một phần tử trong hàng đợi này ngăn cản được thêm vào bộ sưu tập được chỉ đinh.
* ***NullPointerException****:* Nếu bộ sưu tập rỗng.
* ***IllegalArgumentException****:* Nếu bộ sư tập chỉ định là hàng đợi hoặc một số thuộc tính của một phần tử của hàng đợi này ngăn việc thêm vào bộ sưu tập được chỉ định.
* Cú pháp:

/\*\*

\* Phương thức chuyển đổi phần tử của hàng đợi thành tập các phần tử

\* (Collection). Trả về số lượng phần tử đã chuyển đổi. <br>

\* Create by: BVHA (12/05/2021)

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@param** collec - Danh sách để chứa giá trị chuyển đổi

\* **@return**

\*/

**public** **static** <E> **int** drainToCollection(ArrayBlockingQueue<E> abq, Collection<E> collec)

{

**return** abq.drainTo(collec);

}

*Hình 3.25: Phương thức drainTo một tham số*

* *drainTo(Collection c, int maxE)*

Phương thức dùng để chuyển một lượng phần tử cố định tới bộ sưu tập. Sau khi chuyển đổi, hàng đợi chỉ còn tồn tại những phần tử chưa chuyển đổi. Còn trả về giá trị và báo lỗi thì giống với mục *drainTo(Collection)*

* Cú pháp:

/\*\*

\* Phương thức chuyển đổi phần tử của hàng đợi thành tập các phần tử

\* (Collection). Trả về số lượng phần tử đã chuyển đổi. Được giới hạn

\* phần tử chuyển đổi bởi <b>maxElements.</b> Giá trị lấy tuân theo FIFO,

\* nếu maxElements nhở hơn kích thước sẽ lấy đúng số lượng giới hạn,

\* ngượi lại nếu lớn hơn thì sẽ lấy toàn bộ <br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@param** collec - Danh sách để chứa giá trị chuyển đổi

\* **@param** maxElements - Số lượng tối đa được chuyển đổi

\* **@return**

\* **@throws** IOException

\*/

**public** **static** <E> **int** drainToCollection(ArrayBlockingQueue<E> abq, Collection<E> collec, **int** maxElements) **throws** IOException

{

**return** abq.drainTo(collec, maxElements);

}

*Hình 3.26: Phương thức drainTo hai tham số*

### Phương thức toArray(), toArray(T[] arr )

* *toArray()*

Phương thức dùng để tạo một mảng chứa những giá trị giống với hàng đợi hiện tại. Về cơ bản, phương thức thực thi sao chép tất cả phần tử của hàng đợi sang một mảng mới. Có thể coi phương thức này là cầu nối giữa mảng và danh sách.

* Cú pháp:

/\*\*

\* Phương thức trả về một mảng chứa tất cả các giá trị của hàng đợi<br>

\* Create by: BVHA (12/05/2021)

**public** **static** <E> Object toArray(ArrayBlockingQueue<E> abq)

{

Object[] array = abq.toArray();

**return** array;

}

*Hình 2.37: Phương thức toArray hai tham số*

* *toArray(T[] arr)*

Phương thức dùng để tạo một mảng chữa những giá trị giống với hàng đợi hiện tại. Phương thức sẽ thêm một điều kiện bổ sung. Kiểu mảng trả về giống với mảng chỉ định được truyền vào, nếu kích thước hàng đợi nhỏ hơn hoặc bằng với mảng chỉ định. Nếu không một mảng mới được cấp phát với kiểu giống với mảng được chỉ định và kích thước của mảng bằng với kích thước của hàng đợi. Phương thức này được coi như cầu nối giữa mảng và bộ sưu tập.

* Cú pháp:

/\*\*

\* Phương thức trả về một mảng chứ tất cả các giá trị của hàng đợi. The

\* <i>arr</i> sẽ chứa tất cả các phần tử của hàng đợi, nếu mảng có kích

\* thước lớn hơn hoặc bằng kích thước hàng đợi<br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq - Hàng đợi

\* **@param** arr - Mảng kiểu dữ liệu

\* **@return**

\*/

**public** **static** <E> Object toArray(ArrayBlockingQueue<E> abq, E[] arr)

{

// Tạo một mảng đối tượng, với giá trị lấy từ hàng đợi

Object[] array = abq.toArray(arr);

**return** array;

}

*Hình 3.28: Phương thúc toArray một tham số*

### Phương thức *remainingEmpty()*

* Phương thức trả về số lượng phần tử có thể được thêm vào hàng đợi mà không bị chặn. Có thể hiểu là giá trị trả về là kích thước tổng trừ đi số lượng phần tử hàng đợi đang chứa.
* Cú pháp:

/\*\*

\* Phương thức trả về số lượng phần tử có thể lưu trữ<br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq

\* **@return**

\*/

**public** **static** <E> **int** getCapacityEmpty(ArrayBlockingQueue<E> abq)

{

**return** abq.remainingCapacity();

}

*Hình 3.29: Phương thức remainingEmpty*

### Phương thức spliterator()

* Phương thức trả về *Spliterator* trên các phân tử của hàng đợi. *Spliterator* có thể được sử dụng với Streams trong Java 8. *Spliterator* có thể duyệt các phần tử riêng lẻ và hàng loạt.
* Cú pháp:

/\*\*

\* Phương thức trả về Spliterator dựa vào các phần tử trong hàng đợi<br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq

\* **@return**

\*/

**public** **static** <E> Spliterator getSpliteratorList(ArrayBlockingQueue<E> abq )

{

Spliterator<E> spliterators = abq.spliterator();

**return** spliterators;

}

*Hình 3.30: Phương thức spliteratior*

### Phương thức *forEach()*

* Phương thức thực thi duyệt qua từng phần tử của danh sách, hàng đợi.
* Cú pháp:

/\*\*

\* Hiển thị lần lượt các phần tử trong hàng đợi<br>

\* Create by: BVHA (12/05/2021)

\*

\* **@param** <E>

\* **@param** abq

\*/

**public** **static** <E> **void** printArrayBlockingQueue(ArrayBlockingQueue<E> abq)

{

abq.forEach(e -> System.***out***.println(e));

}

*Hình 3.31: Phương thúc forEach*

## Kịch bản chạy chương trình

### Kịch bản khởi tạo đối tượng

* *Kịch bản chạy*

**public** **static** **void** fn\_constructor() {

// Hàm khởi tạo đối tượng có kích thước = 5

ArrayBlockingQueue<String> abq = MyArrayBlockingQueue.*createArrayBlockingQueue*(5);

// Thêm phần tử

abq.add("A");

abq.add("B");

// Hàm khởi tạo đối tượng có kích thước = 5, true

ArrayBlockingQueue<String> abq1 = MyArrayBlockingQueue.*createArrayBlockingQueue*(5, **true**);

// Thêm phần tử

abq1.add("A");

abq1.add("B");

// Hàm khởi tạo đối tượng có kích thước = 5, true, clo

Collection<String> clo = **new** ArrayList<String>();

// Thêm phần tử

clo.add("A");

clo.add("B");

clo.add("C");

ArrayBlockingQueue<String> abq2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(5, **true**, clo);

// Hiển thị hàng đợi

System.***out***.println("Hàng đợi 1: " + abq);

System.***out***.println("Hàng đợi 2: " + abq1);

System.***out***.println("Hàng đợi 3: " + abq2);

}

*Hình 3.32: Kịch bản khởi tạo đối tượng*

* *Kết quả*

Hàng đợi 1: [A, B]

Hàng đợi 2: [A, B]

Hàng đợi 3: [A, B, C]

*Hình 3.33: Kết quả*

### Kịch bản thêm phần tử

**Phương thức *add(), addAll()***

* *Kịch bản chạy*

**public** **static** **void** fn\_add() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> abq = MyArrayBlockingQueue.*createArrayBlockingQueue*(5);

// Thêm phần tử mới vào hàng đợi

MyArrayBlockingQueue.*addElement*(abq, 4);

// Hiển thị hàng đợi

System.***out***.println("Add: " + abq);

// Khởi tạo và thêm giá trị vào danh sách

Collection<Integer> clo = **new** ArrayList<Integer>();

clo.add(1);

clo.add(3);

clo.add(2);

// Thêm danh sách vào hàng đợi

MyArrayBlockingQueue.*addAllElement*(abq, clo);

// Hiển thị hàng đợi

System.***out***.println("Add all: " + abq);

}

*Hình 3.34: Kịch bản chạy phương thức add-addAll*

* *Kết quả*

Add: [4]

Add all: [4, 1, 3, 2]

*Hình 3.35: Kết quả*

**Phương thức *offer()***

* *Kịch bản chạy*

**public** **static** **void** fn\_addOffer() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

// Thêm phần tử vào hàng đợi

System.***out***.println("Thêm 423: " + MyArrayBlockingQueue.*addElementByOffer*(queue, 423));

System.***out***.println("Thêm 243: " + MyArrayBlockingQueue.*addElementByOffer*(queue, 243));

// Kiểm tra đầy hàng đợi

**if** (MyArrayBlockingQueue.*getCapacityEmpty*(queue) == 0) {

System.***out***.println("Hàng đợi đầy");

System.***out***.println("Hàng đợi tồn tại " + queue);

} **else** {

System.***out***.println("Hàng đợi chưa đầy");

System.***out***.println("Hàng đợi tồn tại" + queue);

}

// Cố gắng thêm phần tử vào

System.***out***.println("Thêm 123: " + MyArrayBlockingQueue.*addElementByOffer*(queue, 123));

System.***out***.println("Thêm 321: " + MyArrayBlockingQueue.*addElementByOffer*(queue, 321));

}

*Hình 3.36: Kich bản chạy phương thúc offer*

* *Kết quả*

Thêm 423: true

Thêm 243: true

Hàng đợi đầy

Hàng đợi tồn tại [423, 243]

Thêm 123: false

Thêm 321: false

*Hình 3.37: Kết quả*

* *Kịch bản chạy*

**public** **static** **void** fn\_addOfferV2() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

System.***out***.println("Thêm 423: " + MyArrayBlockingQueue.*addElementByOffer*(queue, 433, 10, TimeUnit.***SECONDS***));

System.***out***.println("Thêm 456: " + MyArrayBlockingQueue.*addElementByOffer*(queue, 456, 10, TimeUnit.***SECONDS***));

// Kiểm tra đầy hàng đợi

**if** (MyArrayBlockingQueue.*getCapacityEmpty*(queue) == 0) {

System.***out***.println("Hàng đợi đầy");

System.***out***.println("Hàng đợi tồn tại " + queue);

} **else** {

System.***out***.println("Hàng đợi chưa đầy");

System.***out***.println("Hàng đợi tồn tại" + queue);

}

// Tiếp tục thêm nhân viên

System.***out***.println("Thêm 123: " + MyArrayBlockingQueue.*addElementByOffer*(queue, 123, 5, TimeUnit.***SECONDS***));

}

*Hình 3.38: Kịch bản chạy phương thức offer có tham số*

* *Kết quả*

Thêm 423: true

Thêm 456: true

Hàng đợi đầy

Hàng đợi tồn tại [433, 456]

*Hình 3.39: Kết quả*

**Phương thức *put()***

* *Kịch bản chạy*

**public** **static** **void** fn\_addPut() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

// Thêm mới phần tử

MyArrayBlockingQueue.*addElementByPut*(queue, 223);

MyArrayBlockingQueue.*addElementByPut*(queue, 456);

// Hiển thị phần tử

System.***out***.println("Hàng đợi: " + queue);

}

*Hình 3.39: Kịch bản chạy phương thức put*

* *Kết quả*

Hàng đợi: [223, 456]

*Hình 3.40: Kết quả*

### Kịch bản loại bỏ phần tử

**Phương thức *remove()***

* Kịch bản chạy

**public** **static** **void** fn\_removeV2() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

// Hiển thị hàng đợi

System.***out***.println("Hàng đợi 1: " + queue);

System.***out***.println("Loại bỏ giá trị hàng đợi 1: " + MyArrayBlockingQueue.*removeElementFirst*(queue));

System.***out***.println("Hàng đợi 1: " + queue + "\n");

// Hiển thị mảng rỗng

System.***out***.println("Hàng đợi 2: " + queue2);

System.***out***.println("Loại bỏ giá trị hàng đợi 2: " + MyArrayBlockingQueue.*removeElementFirst*(queue2));

}

*Hình 3.41: Kịch bản chạy phương thức remove*

* *Kết quả*

Hàng đợi 1: [123, 456]

Loại bỏ giá trị hàng đợi 1: 123

Hàng đợi 1: [456]

Hàng đợi 2: []

*Hình 3.42: Kết quả*

**Phương thức *removeAll()***

* Kịch bản chạy

**public** **static** **void** fn\_removeAll() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

queue2.add(123);

// Hiển thị hàng đợi

System.***out***.println("Hàng đợi 1: " + queue);

System.***out***.println("Loại bỏ giá trị hàng đợi 1: " + MyArrayBlockingQueue.*removeAllElement*(queue, queue2));

System.***out***.println("Hàng đợi 1: " + queue + "\n");

// Hiển thị mảng rỗng

System.***out***.println("Hàng đợi 2: " + queue2);

System.***out***.println("Loại bỏ giá trị hàng đợi 2: " + MyArrayBlockingQueue.*removeAllElement*(queue2, queue));

}

*Hình 3.43: Kịch bản chạy phương thức removeAll*

* Kết quả

Hàng đợi 1: [123, 456]

Loại bỏ giá trị hàng đợi 1: true

Hàng đợi 1: [456]

Hàng đợi 2: [123]

Loại bỏ giá trị hàng đợi 2: false

*Hình 3.44: Kết quả*

**Phương thức *clear()***

* Kịch bản chạy

**public** **static** **void** fn\_clear() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

queue2.add(123);

// Hiển thị hàng đợi

System.***out***.println("Hàng đợi 1: " + queue);

MyArrayBlockingQueue.*clear*(queue);

System.***out***.println("Hàng đợi 1: " + queue + "\n");

System.***out***.println("Hàng đợi 2: " + queue2);

MyArrayBlockingQueue.*clear*(queue2);

System.***out***.println("Hàng đợi 2: " + queue2);

}

*Hình 3.45: Kịch bản chạy phương thức clear*

* *Kết quả*

Hàng đợi 1: [123, 456]

Hàng đợi 1: []

Hàng đợi 2: [123]

Hàng đợi 2: []

*Hình 3.46: Kết quả*

### Kịch bản kiểm tra tồn phần tử

**Phương thức *contains()***

* *Kịch bản chạy*

**public** **static** **void** fn\_contains() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

// Hiển thị hàng đợi

System.***out***.println("Hàng đợi 1 tồn tại 123: " + MyArrayBlockingQueue.*containElement*(queue, 123));

// Hiển thị mảng rỗng

System.***out***.println("Hàng đợi 2 tồn tại 123: " + MyArrayBlockingQueue.*containElement*(queue2, 123));

}

*Hình 3.47: Kịch bản chạy phương thức contains*

* *Kết quả*

Hàng đợi 1 tồn tại 123: true

Hàng đợi 2 tồn tại 123: false

*Hình 3.48: Kết quả*

**Phương thức *containsAll()***

* *Kịch bản chạy*

**public** **static** **void** fn\_containsAll() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

// Khởi tạo danh sách và thêm phần tử

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

list.add(123);

list.add(456);

// Hiển thị hàng đợi

System.***out***.println("Hàng đợi 1 tồn tại 123: " + MyArrayBlockingQueue.*containAllElement*(queue, list));

// Hiển thị mảng rỗng

System.***out***.println("Hàng đợi 2 tồn tại 123: " + MyArrayBlockingQueue.*containAllElement*(queue2, list));

}

*Hình 3.49: Kịch bản chạy phương thức containsAll*

* *Kết quả*

Hàng đợi 1 tồn tại 123: true

Hàng đợi 2 tồn tại 123: false

*Hình 3.50: Kết quả*

### Kịch bản chuyển đổi danh sách

**Phương thức *drainTo()***

* *Kích bản chạy*

**public** **static** **void** fn\_draintTo() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

System.***out***.println("Trước khi drainTo\n" + queue);

MyArrayBlockingQueue.*drainToCollection*(queue, queue2);

System.***out***.println("Sau khi drainTo\n" + queue);

System.***out***.println("queue2: " + queue2);

}

*Hình 3.51: Kịch bản chạy phương thức drainTo không tham số*

* *Kết quả*

Trước khi drainTo:

[123, 456]

Sau khi drainTo: []

queue2: [123, 456]

*Hình 3.52: Kết quả*

* *Kịch bản chạy*

**public** **static** **void** fn\_drainToV2() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

System.***out***.println("Trước khi drainTo\n" + queue);

**try** {

MyArrayBlockingQueue.*drainToCollection*(queue, queue2, 1);

} **catch** (IOException e) {

e.printStackTrace();

}

System.***out***.println("Sau khi drainTo\n" + queue);

System.***out***.println("queue2: " + queue2);

}

*Hình 3.53: Kịch bản chạy phương thức drainTo có tham số*

* *Kết quả*

Trước khi drainTo

[123, 456]

Sau khi drainTo

[456]

queue2: [123]

*Hình 3.54: Kết quả*

### Kịch bản chuyển đổi mảng

**Phương thức *toArray()***

* *Kịch bản chạy*

**public** **static** **void** fn\_ToArray() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

Object[] array = MyArrayBlockingQueue.*toArray*(queue);

Object[] array2 = MyArrayBlockingQueue.*toArray*(queue2);

System.***out***.println("Mảng 1:");

MyArrayBlockingQueue.*printArray*(array);

System.***out***.println("\nMảng 2:");

MyArrayBlockingQueue.*printArray*(array2);

}

*Hình 3.55: Kịch bản chạy phương thức toArray một tham số*

* *Kết quả*

Mảng 1:

123

456

Mảng 2:

*Hình 3.56: Kết quả*

* *Kịch bản chạy*

**public** **static** **void** fn\_ToArrayV2() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

// Tạo mảng

Integer[] array = **new** Integer[2];

// gọi hàm toArray(T[] a)

Object[] ReturnArray = MyArrayBlockingQueue.*toArray*(queue, array);

System.***out***.println("Hàng đợi: " + queue);

System.***out***.println("\nMảng chuyển từ toArray():");

MyArrayBlockingQueue.*printArray*(array);

System.***out***.println("\nMảng trả về từ toArray():");

MyArrayBlockingQueue.*printArray*(ReturnArray);

}

*Hình 3.57: Kịch bản chạy phương thức toArray hai tham số*

* *Kết quả*

Hàng đợi: [123, 456]

Mảng chuyển từ toArray():

123

456

Mảng trả về từ toArray():

123

456

*Hình 3.58: Kết quả*

### Kịch bản chuyển đổi dạng spliterator

**Phương thức *spliterator()***

* *Kịch bản chạy*

**public** **static** **void** fn\_spliterator() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

Spliterator<Integer> number1 = MyArrayBlockingQueue.*getSpliteratorList*(queue);

Spliterator<Integer> number2 = MyArrayBlockingQueue.*getSpliteratorList*(queue2);

System.***out***.println("Kích thước của number1 : " + number1.estimateSize());

System.***out***.println("Danh sách number1:");

//Dùng hàm forEachRemaining

number1.forEachRemaining((n) -> System.***out***.println(n));

System.***out***.println("Kích thước của number2 : " + number1.estimateSize());

System.***out***.println("Danh sách number2:");

//Dùng hàm forEachRemaining

number2.forEachRemaining((n) -> System.***out***.println(n));

}

*Hình 3.59: Kịch bản chạy phương thức spliterator*

* *Kết quả*

Kích thước của number1 : 2

Danh sách number1:

123

456

Kích thước của number2 : 2

Danh sách number2:

*Hình 3.40: Kết quả*

### Kịch bản lấy kích thước chưa sử dụng

**Phương thức *remainingCapacity()***

* *Kịch bản chạy*

**public** **static** **void** fn\_remainingCapacity() {

// khởi tạo đối tượng

ArrayBlockingQueue<Integer> queue = TestArrayBlockingQueue.*fn\_arrayBlocking*();

ArrayBlockingQueue<Integer> queue2 = MyArrayBlockingQueue.*createArrayBlockingQueue*(2);

System.***out***.println("Hàng đợi 1: " + queue);

System.***out***.println("Kích thước có thể lưu trữ hàng đợi 1: " + MyArrayBlockingQueue.*getCapacityEmpty*(queue));

System.***out***.println("Hàng đợi 2: " + queue2);

System.***out***.println("Kích thước có thể lưu trữ hàng đợi 2: " + MyArrayBlockingQueue.*getCapacityEmpty*(queue2));

}

*Hình 3.41: Kịch bản chạy phương thức remainingCapacity*

* *Kết quả*

Hàng đợi 1: [123, 456]

Kích thước có thể lưu trữ hàng đợi 1: 0

Hàng đợi 2: []

Kích thước có thể lưu trữ hàng đợi 2: 2

# ConcurrentLinkedQueue

## Khái niệm chung về ConcurrentLinkedQueue

Lớp ConcurrentLinkedQueue trong Java là một phần của Java Collection Framework. Nó thuộc về gói java.util.concurrent. Nó đã được giới thiệu trong JDK 1.5. Nó được sử dụng để implement Queue với sự trợ giúp của LinkedList. Đây là một triển khai hàng đợi an toàn theo luồng không bị ràng buộc, nó sẽ chèn các phần tử vào cuối “ hàng đợi ” theo kiểu FIFO (nhập trước xuất trước). Nó có thể được sử dụng khi một “ hàng đợi ” không bị ràng buộc được chia sẻ giữa nhiều luồng. Lớp này không cho phép các phần tử rỗng. Các trình lặp có tính nhất quán yếu. Lớp này và iterator của nó thực thi tất cả các phương thức tùy chọn của giao diện Queue và Iterator.

## ConcurrentLinkedQueue trong java

### Cách khai báo trong ConcurrentLinkedQueue

**public** **class** ConcurrentLinkedQueue<E> **extends** AbstractQueue<E>

**implements** Queue<E>, java.io.Serializable {

// methods ...

}

*Hình 4.1: Khai báo hàm khởi tạo*

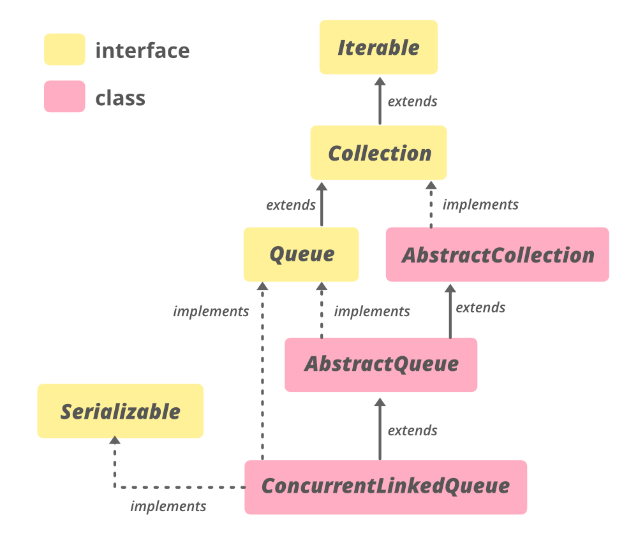
### Hệ thống phân cấp lớp

java.lang.Object

↳ java.util.AbstractCollection<E>

↳ java.util.AbstractQueue<E>

↳ Class ConcurrentLinkedQueue<E>



*Hình 4.2: Mô hình phân cấp lớp hệ thống của ConcurrentLinkedQueue*

### Cách tạo một ConcurrentLinkedQueue

* Constructor dùng để khởi tạo một hàng đợi trống:

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>();

* Constructor dùng để khởi tạo một hàng đợi với những phần tử được truyền vào dưới dạng các tham số:

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>(Collection<E> c);

### Thêm phần tử

* Thêm một phần tử: phương thức add () hoặc offer() nó chèn phần tử, được truyền dưới dạng tham số ở đuôi ConcurrentLinkedQueue này.

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>();

clq.add(e);

// or

clq.offer(e);

* Phương thức này trả về True nếu chèn thành công. ConcurrentLinkedQueue là không bị ràng buộc, vì vậy phương thức này sẽ không bao giờ ném IllegalStateException hoặc trả về false.

*Hình 4.3: Thêm phần tử add*

* Thêm nhiều phần tử: addAll () Nó chèn tất cả các phần tử của Collection, được truyền dưới dạng tham số ở cuối ConcurrentLinkedQueue. Việc chèn phần tử theo thứ tự như được trả về bởi trình vòng lặp của Collection.

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>();

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

clq.addAll(list);

*Hình 4.4: Thêm phần tử addAll*

### Lấy giá trị

* Hiển thị phần tử đầu tiên sử dụng phương thức peek(). Phương thức này lấy phần tử đầu tiên và không xóa phần tử đó ra khỏi hàng đợi.

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>();

clq.peek();

*Hình 4.5: Lấy giá trị bằng phương thức peek*

* Phương thức element() hiển thị phần tử đầu tiên trong hàng đợi nhưng không xóa .

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>();

clq.element();

*Hình 4.6: Lấy giá trị bằng phương thức element*

* Hiển thị phần tử đầu tiên sử dụng phương thức poll(). Phương thức này xóa và hiển thị phần tử đầu tiên.

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>();

clq.poll();

*Hình 4.7: Lấy giá trị bằng phương thức poll*

1. Xóa giá trị

* Phương thức remove (Object o) của ConcurrentLinkedQueue được sử dụng để loại bỏ một trường hợp duy nhất của phần tử được chỉ định nếu nó có mặt. Nó loại bỏ một phần tử e sao cho o.equals (e). Nó trả về true nếu ConcurrentLinkedQueue này chứa phần tử được chỉ định khác, nó sẽ trả về false.

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<>();

Object object = **new** Object();

clq.remove(object);

*Hình 4.8: Xóa giá trị bằng phương thức remove*

* Phương thức iterator () của ConcurrentLinkedQueue được sử dụng để trả về một trình lặp của các phần tử giống như ConcurrentLinkedQueue này theo một trình tự thích hợp. Các phần tử được trả về từ phương thức này chứa các phần tử theo thứ tự từ đầu tiên (đầu) đến cuối cùng (đuôi). Trình lặp được trả về là nhất quán yếu.

ConcurrentLinkedQueue<E> clq = **new** ConcurrentLinkedQueue<E>();

// Call iterator() method

Iterator<E> iterator = clq.iterator();

// Print elements of iterator

**while** (iterator.hasNext()) {

// value = iterator.next();

}

*Hình 4.9: Phương thức iterator*

## Các phương thức của ConcurrentLinkedQueue

### Phương thức add​(E e) .

* Phương thức add () của ConcurrentLinkedQueue được sử dụng để chèn phần tử, được truyền dưới dạng tham số để add () của ConcurrentLinkedQueue, ở phần đuôi của ConcurrentLinkedQueue này. Phương thức này trả về True nếu chèn thành công. ConcurrentLinkedQueue là không bị ràng buộc, vì vậy phương thức này sẽ không bao giờ ném IllegalStateException hoặc trả về false.
* Cú pháp:

/\*\*

\* - Phương thức <b>thêm</b> một chuỗi str vào hàng đợi queue.</br>

\* - Phương thức này sẽ chèn một chuỗi str vào cuối của hàng đợi.</br>

\* - Phương thức này sẽ trả về true nếu thêm thành công và nó sẽ trả về

\* false nếu thêm thất bại.

\*

\* **@param** queue

\* **@param** str

\* **@return**

\*/

**public** **static** **boolean** add(ConcurrentLinkedQueue<String> queue, String str)

{

// Phương thức add() sẽ chèn phần tử được chỉ định vào cuối của hàng // đợi

**return** queue.add(str);

}

*Hình 4.10: Phương thức add*

### Phương thức addAll​(Collection<? extends E> c) .

* Phương thức addAll () của ConcurrentLinkedQueue được sử dụng để chèn tất cả các phần tử của Collection, được truyền dưới dạng tham số cho phương thức này, vào cuối ConcurrentLinkedQueue. Việc chèn phần tử theo thứ tự như được trả về bởi collections iterator.

### Phương thức thêm từ danh sách ArrayList.

/\*\*

\* - Phương thức <b>thêm</b> một danh sách chuỗi vào hàng đợi.</br>

\* - Phương thức này sẽ chèn một danh sách String vào cuối của hàng

\* đợi.</br>

\* - Phương thức này sẽ trả về true nếu thêm thành công và nó sẽ trả

\* về false nếu thêm thất bại..

\*

\* **@param** queue

\* **@param** listString

\* **@return**

\*/

**public** **static** **boolean** addAll(ConcurrentLinkedQueue<String> queue,

ArrayList<String> listString) {

// Phương thức addAll() sẽ chèn tất cả các phần tử của danh sách

// vào cuối của hàng đợi.

**return** queue.addAll(listString);

}

*Hình 4.11: Phương thức thêm từ danh sách*

### Phương thức thêm từ ConcurrentLinkedQueue.

/\*\*

\* - Phương thức <b>thêm</b> một hàng đợi newQueue vào một hàng đợi khác

\* queue.</br>

\* - Phương thức này sẽ chèn một hàng đợi newQueue vào cuối của hàng đợi

\* queue.</br>

\* - Phương thức này sẽ trả về true nếu thêm thành công và nó sẽ trả về

\* false nếu thêm thất bại.

\*

\* **@param** queue

\* **@param** newQueue

\* **@return**

\*/

**public** **static** **boolean** addAll(ConcurrentLinkedQueue<String> queue,

ConcurrentLinkedQueue<String> newQueue) {

// Phương thức addAll() sẽ chèn tất cả các phần tử của newQueue vào

// cuối của hàng đợi queue.

**return** queue.addAll(newQueue);

}

*Hình 4.12: Phương thức thêm từ ConcurrentLinkedQueue*

1. Phương thức contains​(Object o) .

* Phương thức contains() của ConcurrentLinkedQueue trả về true nếu ConcurrentLinkedQueue chứa đối tượng o, được truyền dưới dạng tham số. Phương thức này trả về true nếu và chỉ khi ConcurrentLinkedQueue này chứa ít nhất một phần tử e bằng đối tượng o được truyền dưới dạng tham số, tức là o.equals (e).
* Cú pháp:

/\*\*

\* Phương thức tìm kiếm phần tử trong hàng đợi sử dụng phương thức

\* queue.

\*

\* **@param** queue

\* **@param** str

\* **@return**

\*/

**public** **static** **boolean** search(ConcurrentLinkedQueue<String> queue, String str)

{

// Phương thức contains() sẽ trả về true nếu hàng đợi chứa phần tử được chỉ

// định.

**return** queue.contains(str);

}

*Hình 4.13: Phương thức contains*

### Phương thức forEach​(Consumer<? super E> action).

* Đây là phương thức dùng để đọc thông tin của từng phần tử trong hàng đợi.
* Cú pháp:

/\*\*

\* Phương thức in hàng đợi ra màn hình sử dụng forEach().

\* **@param** queue

\*/

**public** **static** **void** printQueueByForEach(ConcurrentLinkedQueue<String> queue) {

queue.forEach(element->{

// In từng phần tử trong hàng đợi ra màn hình

System.***out***.println(element);

});

}

*Hình 4.14: Phương thức forEach*

### Phương thức isEmpty() .

* Phương thức isEmpty () của ConcurrentLinkedQueue được sử dụng để kiểm tra xem hàng đợi này có trống hay không. Nó trả về true nếu ConcurrentLinkedQueue chứa số phần tử bằng không có nghĩa là nếu ConcurrentLinkedQueue trống.
* Cú pháp:

/\*\*

\* Phương thức kiểm tra xem hàng đợi có rỗng không</br>

\* -> Trả về true nếu hàng đợi không chứa phần tử.

\* **@param** queue

\* **@return**

\*/

**public** **static** **boolean** isEmpty(ConcurrentLinkedQueue<String> queue)

{

**return** queue.isEmpty();

}

*Hình 4.15: Phương thức isEmpty*

### Phương thức iterator().

* Phương thức iterator () của ConcurrentLinkedQueue được sử dụng để trả về một trình lặp của các phần tử giống như ConcurrentLinkedQueue này theo một trình tự thích hợp. Các phần tử được trả về từ phương thức này chứa các phần tử theo thứ tự từ đầu tiên (đầu) đến cuối cùng (đuôi). Trình lặp được trả về là nhất quán yếu.
* Cú pháp:

/\*\*

\* Phương thức in từng phần tử trong queue ra màn hình sử dụng iterator().

\* **@param** queue

\*/

**public** **static** **void** printQueueByIterator(ConcurrentLinkedQueue<String> queue)

{

// Call iterator() method

Iterator<String> iterator = queue.iterator();

// Print elements of iterator

System.***out***.println("\nThe String Values of iterator are:");

**while** (iterator.hasNext()) {

System.***out***.println(iterator.next());

}

}

*Hình 4.16: Phương thức iterator*

### Phương thức offer​(E e) .

* Phương thức offer () của ConcurrentLinkedQueue được sử dụng để chèn phần tử, được truyền dưới dạng tham số, vào đuôi ConcurrentLinkedQueue này. Phương thức này trả về True nếu chèn thành công. ConcurrentLinkedQueue là không bị ràng buộc, vì vậy phương thức này offer () sẽ không bao giờ trả về false.
* Cú pháp:

/\*\*

\* - Phương thức <b>thêm</b> một chuỗi str vào hàng đợi queue.</br>

\* - Phương thức này sẽ chèn một chuỗi str vào cuối của hàng đợi.</br>

\* - Phương thức này sẽ trả về true nếu thêm thành công và nó sẽ trả về

\* false nếu thêm thất bại.

\* **@param** queue

\* **@param** str

\* **@return**

\*/

**public** **static** **boolean** addByOffer(ConcurrentLinkedQueue<String> queue, String str)

{

// Phương thức offer() sẽ chèn phần tử được chỉ định vào cuối của // hàng đợi

**return** queue.offer(str);

}

*Hình 4.17: Phương thức offer*

### Phương thức remove​(Object o) .

* Phương thức remove (Object o) của ConcurrentLinkedQueue được sử dụng để loại bỏ một trường hợp duy nhất của phần tử được chỉ định khỏi ConcurrentLinkedQueue này, nếu nó có mặt. Phương thức này loại bỏ một phần tử e sao cho o.equals (e) nếu ConcurrentLinkedQueue này chứa một hoặc nhiều phần tử như vậy. Phương thức Remove () trả về true nếu ConcurrentLinkedQueue này chứa phần tử được chỉ định nếu không nó sẽ trả về false.
* Cú pháp:

/\*\*

\* Phương thức xóa phần tử có giá trị value trong hàng đợi queue.

\* **@param** queue

\* **@param** value

\* **@return**

\*/

**public** **static** **boolean** removeElement(ConcurrentLinkedQueue<String> queue, String value)

{

**return** queue.remove(value);

}

*Hình 4.18: Phương thức remove*

### Phương thức removeAll​(Collection<?> c) .

* Loại bỏ tất cả các phần tử của Collection này cũng có trong Collection được chỉ định (thao tác tùy chọn).
* Phương thức loại bỏ tất cả các phần tử trong queue theo ArrayList.

/\*\*

\* Phương thức xóa tất cả các phần tử của hàng đợi queue có trong danh

\* sách listRemove.

\* **@param** queue

\* **@param** listRemove

\* **@return**

\*/

**public** **static** **boolean** removeAllByList(ConcurrentLinkedQueue<String> queue, ArrayList<String> listRemove)

{

**return** queue.removeAll(listRemove);

}

*Hình 4.19: Phương thức loại bỏ tất cả phần tử theo arrayList*

* Phương thức loại bỏ tất cả các phần tử trong queue theo một hàng đợi cho trước:

/\*\*

\* Phương thức xóa tất các các phần tử của ngăn xếp queue có trong ngăn

\* xếp queueRemove.

\* **@param** queue

\* **@param** queueRemove

\* **@return**

\*/

**public** **static** **boolean** removeAllByQueue(ConcurrentLinkedQueue<String> queue, ConcurrentLinkedQueue<String> queueRemove)

{

**return** queue.removeAll(queueRemove);

}

*Hình 4.20: Phương loại bỏ tất cả các phần tử theo hàng đợi*

### Phương thức removeIf​(Predicate<? super E> filter) .

* Loại bỏ tất cả các phần tử của tập hợp này thỏa mãn vị từ đã cho.

/\*\*

\* Phương thức xóa giá trị trong hàng đợi với điều kiện xác định.

\* **@param** queue

\* **@param** valueRemove

\* **@return**

\*/

**public** **static** **boolean** removeIf(ConcurrentLinkedQueue<String> queue,String valueRemove)

{

// Phương thức xóa các phần tử trong hàng đợi với điều kiện.

**return** queue.removeIf(value -> value.equals(valueRemove));

}

*Hình 4.21: Phương thức removeIf*

### Phương thức spliterator() .

* Phương thức spliterator () của ConcurrentLinkedQueue được sử dụng để lấy một Spliterator có các phần tử giống như ConcurrentLinkedQueue. Đã tạo Spliterator nhất quán yếu. Nó có thể được sử dụng với Streams trong Java 8. Ngoài ra, nó cũng có thể duyệt các phần tử riêng lẻ và hàng loạt. Spliterator là cách tốt hơn để duyệt qua phần tử vì nó cung cấp nhiều quyền kiểm soát hơn trên các phần tử.

/\*\*

\* Phương thức in từng phần tử trong queue ra màn hình sử dụng

\* spliterator().

\*

\* **@param** queue

\*/

**Public** **static** **void** printQueueBySpliterator(ConcurrentLinkedQueue<String> queue)

{

// create Spliterator of ConcurrentLinkedQueue

// using spliterator() method

Spliterator<String> spt = queue.spliterator();

// print result from Spliterator

System.***out***.println("List of ConcurrentLinkedQueue:");

// forEachRemaining method of Spliterator

spt.forEachRemaining((n) -> System.***out***.println(n + ", "));

}

*Hình 4.22: Phương thức spliterator*

### Phương thức toArray() .

* Phương thức toArray () của ConcurrentLinkedQueue được sử dụng để trả về một mảng có các phần tử giống như của ConcurrentLinkedQueue theo trình tự thích hợp. Về cơ bản, nó sao chép tất cả phần tử từ ConcurrentLinkedQueue sang một mảng mới. Phương thức này hoạt động như một cầu nối giữa mảng và ConcurrentLinkedQueue.
* Cú pháp:

/\*\*

\* Phương thức chuyển hàng đợi thành mảng một chiều sử dụng phương thức

\* toArray();

\* **@param** queue

\* **@return**

\*/

**public** **static** Object [] toArray(ConcurrentLinkedQueue<String> queue){

// Phương thức toArray() dùng để chuyển một hàng đợi thành mảng đối

// tượng.

Object[] obj = queue.toArray();

**return** obj;

}

*Hình 4.23: Phương thức toArray*

### Phương thức peek()

* Phương thưc peek() dùng để lấy phần tử đầu tiên trong hàng đợi và không xóa phần tử đó ra khỏ hàng đợi.
* Cú pháp:

/\*\*

\* Phương thức lấy phần tử đầu tiên trong hàng đợi sử dụng phương thức

\* peek().

\* **@param** queue

\* **@return**

\*/

**public** **static** String getValueHeadByPeek(ConcurrentLinkedQueue<String> queue) {

// Phương thức peek() dùng để lấy phần tử đầu tiên trong hàng đợi.

// Không xóa phần tử đó ra khỏi hàng đợi.

**return** queue.peek();

}

*Hình 4.24: Phương thức peek*

### Phương thức element()

* Phương thức element() dùng để lấy phần tử đầu tiên trong hàng đợi và không xóa phần tử đó ra khỏi hàng đợi.
* Cú pháp:

/\*\*

\* Phương thức lấy phần tử đầu tiên trong hàng đợi sử dụng phương thức

\* element().

\* **@param** queue

\* **@return**

\*/

**public** **static** String

getValueHeadByElement(ConcurrentLinkedQueue<String> queue)

{

// Phương thức element() dùng để lấy phần tử đầu tiên trong hàng // đợi.

// Không xóa phần tử đó ra khỏi hàng đợi.

**return** queue.element();

}

*Hình 4.25: Phương thức element*

### Phương thức poll().

* Phương thức poll() dùng để lấy phần tử đầu tiên trong hàng đợi và xóa nó ra khỏi hàng đợi.
* Cú pháp:

/\*\*

\* Phương thức lấy phần tử đầu tiên trong hàng đợi và xóa giá trị đó ra

\* khỏi hàng đợi sử dụng phương thức poll().

\* **@param** queue

\* **@return**

\*/

**public** **static** String getValueHeadByPoll(ConcurrentLinkedQueue<String>q)

{

// Phương thức poll() dùng để lấy phần tử đầu tiên trong hàng đợi.

// Và xóa phần tử đó ra khỏi hàng đợi.

**return** q.poll();

}

*Hình 4.26: Phương thức poll*

## Kịch bản chạy và kết quả

### Kịch bản chạy cho phương thức khởi tạo với constructor

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Create a ConcurrentLinkedQueue

// using ConcurrentLinkedQueue() contructor

ConcurrentLinkedQueue<String> clq = **new**

ConcurrentLinkedQueue<String>();

clq.add("A");

clq.add("B");

clq.add("C");

clq.add("D");

// Displaying the existing LinkedQueue

System.***out***.println("ConcurrentLinkedQueue: " + clq);

// Create a ConcurrentLinkedQueue

// using ConcurrentLinkedQueue(Collection c)

// constructor

ConcurrentLinkedQueue<String> clq1 = **new**

ConcurrentLinkedQueue<String>(clq);

// Displaying the existing LinkedQueue

System.***out***.println("ConcurrentLinkedQueue1: " + clq1);

}

*Hình 4.27: Kịch bản chạy phương thức khởi tạo*

* Kết quả

ConcurrentLinkedQueue: [A, B, C, D]

ConcurrentLinkedQueue1: [A, B, C, D]

*Hình 4.28: Kết quả*

### Kịch bản chạy cho phương thức add()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq = **new**

ConcurrentLinkedQueue<String>();

//Sử dụng phương thức add

MyConcurrentLinkedQueue.*add*(clq, "A");

// Hiển thị ConcurrentLinkedQueue hiện có

System.***out***.println("ConcurrentLinkedQueue: " + clq);

}

*Hình 4.29: Kịch bản chạy phương thức add*

* Kết quả

ConcurrentLinkedQueue: [A]

*Hình 4.30: Kết quả*

### Kịch bản chạy cho phương thức addAll()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq = **new**

ConcurrentLinkedQueue<String>();

// Sử dụng phương thức addAll() theo ArrayList();

// Tạo một ArrayList

ArrayList<String> temp = **new** ArrayList<String>(List.*of*("A", "B", C"));

// Phương thức addAll từ một ArrayList

MyConcurrentLinkedQueue.*addAll*(clq, temp);

// Hiển thị ConcurrentLinkedQueue hiện có

System.***out***.println("ConcurrentLinkedQueue: " + clq);

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq2 = **new**

ConcurrentLinkedQueue<String>();

// Phương thức addAll từ một ConcurrentLinkedQueue

MyConcurrentLinkedQueue.*addAll*(clq2, clq);

System.***out***.println("ConcurrentLinkedQueue2: "+ clq2);

}

*Hình 4.31: Kịch bản chạy phương thức addAll*

* Kết quả

ConcurrentLinkedQueue: [A, B, C]

ConcurrentLinkedQueue2 [A, B, C]

*Hình 4.32: Kết quả*

### Kịch bản chạy cho phương thức [contains​​](https://www.google.com/url?client=internal-element-cse&cx=009682134359037907028:tj6eafkv_be&q=https://www.geeksforgeeks.org/concurrentlinkedqueue-contains-method-in-java/&sa=U&ved=2ahUKEwjvltatsvLrAhUoxTgGHba8BRkQFjAGegQIAhAC&usg=AOvVaw2qDY3VWUvoPmZro4X5mznF)()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// Hiển thị ConcurrentLinkedQueue hiện có

System.***out***.println(clq);

// Dùng phương thức contains() để tìm kiếm phần tử trong hàng đợi

**if**(MyConcurrentLinkedQueue.*search*(clq, "One")) {

System.***out***.println(**true**);

}**else** {

System.***out***.println(**false**);

}

}

*Hình 4.33: Kịch bản chạy phương thức contains*

* Kết quả:

[One, Two, Three]

true

*Hình 4.34: Kết quả*

### Kịch bản chạy cho phương thức forEach()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// Phương thức in thông tin của các phần tử trong clq ra màn hình

// sử dụng forEach

MyConcurrentLinkedQueue.*printQueueByForEach*(clq);

}

*Hình 4.35: Kịch bản chạy phương thức forEach*

* Kết quả:

One

Two

Three

*Hình 4.36: Kết quả*

### Kịch bản chạy cho phương thức isEmpty()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq = **new**

ConcurrentLinkedQueue<String>();

// Kiểm tra xem hàng đợi trên có rỗng hay không sử dụng phương thức

// isEmpty().

// Nếu rỗng thì trả về true và ngước lại.

**if**(MyConcurrentLinkedQueue.*isEmpty*(clq)) {

System.***out***.println(**true**);

}**else** {

System.***out***.println(**false**);

}

}

*Hình 4.37: Kịch bản chạy phương thức isEmpty*

* Kết quả:

true

*Hình 4.38: Kết quả*

### Kịch bản chạy cho phương thức iterator()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// In clq sử dụng interator()

MyConcurrentLinkedQueue.*printQueueByIterator*(clq);

}

*Hình 4.39: Kịch bản chạy phương thức iterator*

* Kết quả:

The String Values of iterator are:

One

Two

Three

*Hình 4.40: Kết quả*

### Kịch bản chạy cho phương thức offer()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq = **new**

ConcurrentLinkedQueue<String>();

// Sử dụng offer() để thêm phần tử vào clq

MyConcurrentLinkedQueue.*addByOffer*(clq, "Lưu Bá Minh");

// Hiển thị hàng đợi hiện có

System.***out***.println(clq);

}

*Hình 4.41: Kịch bản chạy phương thức offer*

* Kết quả:

[Lưu Bá Minh]

*Hình 4.42: Kết quả*

### Kịch bản chạy cho phương thức remove()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// Hiển thị hàng đợi hiện có

System.***out***.println(clq);

// Xóa phần tử khỏi hàng đợi

MyConcurrentLinkedQueue.*removeElement*(clq, "Two");

// Hiển thị hàng đợi sau khi đã xóa một phần tử

System.***out***.println("Removed : "+clq);

}

*Hình 4.43: Kịch bản chạy phương thức remove*

* Kết quả:

[One, Two, Three]

Removed: [One, Three]

Hình 4.44: Kết quả

### Kịch bản chạy cho phương thức removeAll()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// Hiển thị hàng đợi hiện có

System.***out***.println(clq);

// Tạo một danh sách cần xóa

ArrayList<String> listRemove = **new**

ArrayList<String>(List.*of*("One","Three"));

// Dùng removeAll() để xóa danh sách listRemove ra khỏi hàng đợi clq

MyConcurrentLinkedQueue.*removeAllByList*(clq, listRemove);

// Hiển thị hàng đợi sau khi xóa

System.***out***.println("Removed : "+ clq);

}

*Hình 4.45: Kịch bản chạy phương thức removeAll*

* Kết quả:

[One, Two, Three]

Removed: [Two]

*Hình 4.46: Kết quả*

### Kịch bản chạy cho phương thức removeIf().

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// Hiển thị hàng đợi hiện có

System.***out***.println(clq);

// Tạo một danh sách cần xóa

// Dùng removeIf() để xóa danh sách giá ra khỏi hàng đợi clq

MyConcurrentLinkedQueue.*removeIf*(clq, "Two");

// Hiển thị hàng đợi sau khi xóa

System.***out***.println("Removed : "+ clq);

}

*Hình 4.47: Kịch bản chạy phương thức removeIf*

* Kết quả:

[One, Two, Three]

Removed: [One, Three]

*Hình 4.48: Kết quả*

### Kịch bản chạy cho phương thức spliterator()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// Sử dụng Spliterator để in hàng đợi ra màn hình

MyConcurrentLinkedQueue.*printQueueBySpliterator*(clq);

}

*Hình 4.49: Kịch bản chạy phương thức spliterator*

* Kết quả:

List of ConcurrentLinkedQueue:

One,

Two,

Three

*Hình 4.50: Kết quả*

### Kịch bản chạy cho phương thức toArray()

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

// Hiển thị hàng đợi hiện có

System.***out***.println("ConcurrentLinkedQueue: "+clq);

//Chuyển hàng đợi thành mảng một chiều

Object[] strArray = MyConcurrentLinkedQueue.*toArray*(clq);

// Hiển thị mảng một chiều

System.***out***.println("Array: ");

**for** (Object str : strArray) {

System.***out***.println(str);

}

}

*Hình 4.51: Kịch bản chạy phương thức toArray*

* Kết quả:

ConcurrentLinkedQueue: [One, Two, Three]

Array:

One,

Two,

Three

*Hình 4.52: Kết quả*

### Kịch bản chạy cho phương thức lấy phần tử trong ConcurrentLinkedQueue

* Kịch bản chạy

**public** **static** **void** main(String[] args) {

// Tạo một ConcurrentLinkedQueue

ConcurrentLinkedQueue<String> clq =

MyConcurrentLinkedQueue.*createConcurrentLinkedQueue*();

System.***out***.println("ConcurrentLinkedQueue: " + clq);

String value = "";

// Sử dung peek()

value = MyConcurrentLinkedQueue.*getValueHeadByPeek*(clq);

System.***out***.println("peek() = "+value);

System.***out***.println("ConcurrentLinkedQueue after use peek(): " +clq);

// Sử dung element()

value = MyConcurrentLinkedQueue.*getValueHeadByElement*(clq);

System.***out***.println("element() = "+value);

System.***out***.println("ConcurrentLinkedQueue after use element(): " +clq);

// Sử dung peek()

value = MyConcurrentLinkedQueue.*getValueHeadByPoll*(clq);

System.***out***.println("poll() = "+value);

System.***out***.println("ConcurrentLinkedQueue after use poll() : " +clq);

}

*Hình 4.53: Kịch bản chạy phương thức lấy phần tử trong hàng đợi*

* Kết quả:

ConcurrentLinkedQueue: [One, Two, Three]

peek() = One

ConcurrentLinkedQueue after use peek(): [One, Two, Three]

element() = One

ConcurrentLinkedQueue after use element(): [One, Two, Three]

Poll() = One

ConcurrentLinkedQueue after use poll(): [Two, Three]

*Hình 4.54: Kết quả*

Tài liệu tham khảo:

1. <https://www.geeksforgeeks.org/arrayblockingqueue-class-in-java/>
2. <https://www.geeksforgeeks.org/abstractqueue-in-java-with-examples/>
3. https://www.geeksforgeeks.org/concurrentlinkedqueue-in-java-with-examples/
4. <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html>
5. <https://cafedev.vn/tu-hoc-java-arrayblockingqueue-trong-java/>