Java Queue – Queue in Java

APRIL 2, 2018 BY RAMBABU POSA — 1 COMMENT

Java Queue is an interface available in java.util package and extends java.util.Collection interface.
Just like Java List, Java Queue is a collection of ordered elements (Or objects) but it performs insert and remove operations differently. We can use Queue to store elements before processing those elements.

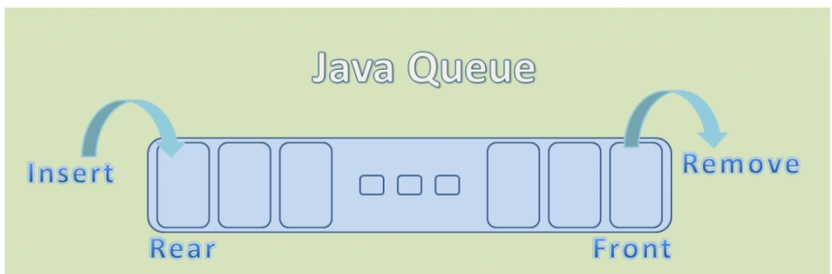


Table of Contents [hide]

2 Java Queue Class Diagram
3 Java Queue Methods
4 Java Queue Basics
5 Java Array to Queue
6 Java Queue to Array
7 Java Queue Common Operations
8 Java Queue Insert Operations
9 Queue add() operation
10 Queue offer() operation
11 Java Queue Delete Operations
12 Queue remove() operation
13 Queue poll() operation
14 Java Queue Examine Operations
15 Queue element() operation
16 Queue peek() operation
17 Java Queue Categories
18 BlockingQueue Operations

Java Queue

In this section, we will discuss some of the important points about Java Queue:

- java.util.Queue interface is a subtype of java.util.Collection interface.
- Just like a real-world queue (for instance, in a bank or at ATM), Queue inserts elements at the end of the queue and removes from the beginning of the queue.
- Java Queue represents an ordered list of elements.
- Java Queue follows FIFO order to insert and remove it's elements. FIFO stands for First In First Out.
- Java Queue supports all methods of Collection interface.
- Most frequently used Queue implementations are LinkedList, ArrayBlockingQueue and PriorityQueue

Java Arrays Annotation and Enum
Java Collections Java IO Operations
Java Exception Handling
MultiThreading and Concurrency
Regular Expressions Advanced Java Concepts

RECOMMENDED TUTORIALS

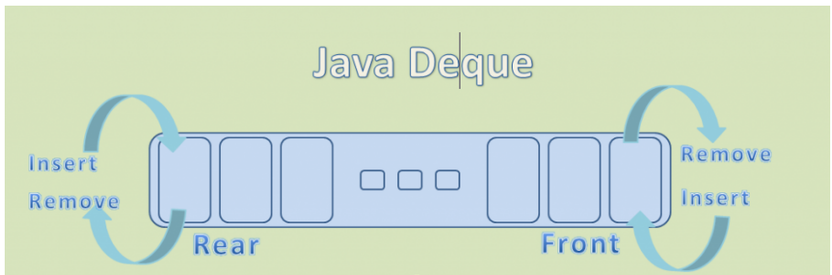
Java Tutorials

- Java IO
- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial

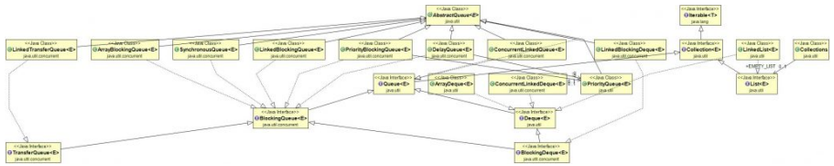
- BlockingQueues do not accept null elements. If we perform any null related operation, it throws NullPointerException.
- BlockingQueues are used to implement Producer/Consumer based applications.
- BlockingQueues are thread-safe.
- All Queues which are available in java.util package are Unbounded Queues and Queues which are available in java.util.concurrent package are Bounded Queues.
- All Deques are not thread-safe.
- ConcurrentLinkedQueue is an unbounded thread-safe Queue based on linked nodes.
- All Queues supports insertion at the tail of the queue and removal at the head of the queue, except Deques.
- Deques are queues but they support element insertion and removal at both ends.



Java Queue Class Diagram

Java Queue interface extends Collection interface. Collection interface extends Iterable interface. Some of the frequently used Queue implementation classes are LinkedList, PriorityQueue, ArrayBlockingQueue, DelayQueue, LinkedBlockingQueue, PriorityBlockingQueue etc.. AbstractQueue provides a skeletal implementation of the Queue interface to reduce the effort in implementing Queue.

› Apache Axis 2
› JAX-RS
› Memcached Tutorial



Java Queue Methods

In this section we will discuss some of the useful and frequently used Java Queue methods:

1. int size(): to get the number of elements in the Set.
2. boolean isEmpty(): to check if Set is empty or not.
3. boolean contains(Object o): Returns true if this Set contains the specified element.
4. Iterator iterator(): Returns an iterator over the elements in this set. The elements are returned in no particular order.
5. boolean removeAll(Collection c): Removes from this set all of its elements that are contained in the specified collection (optional operation).
6. boolean retainAll(Collection c): Retains only the elements in this set that are contained in the specified collection (optional operation).
7. void clear(): Removes all the elements from the set.
8. E remove(): Retrieves and removes the head of this queue.
9. E poll(): Retrieves and removes the head of this queue, or returns null if this queue is empty.
10. E peek(): Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
11. boolean offer(E e): Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
12. E element(): Retrieves, but does not remove, the head of this queue.
13. boolean add(E e): Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.

14. Object[] toArray(): Returns an array containing all of the elements in this set. If this set makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

Java Queue Basics

As Java Queue extends Java Collection, it also supports all Collection interface operations. Let's explore some simple operations in the following example:

```
package com.journaldev.queue;
import java.util.*;

public class QueueExample {
   public static void main(String[] args) {

        Queue<String> queue = new LinkedList<>();
        queue.add("one");
        queue.add("two");
        queue.add("three");
        queue.add("four");
        System.out.println(queue);

        queue.remove("three");
        System.out.println(queue);
        System.out.println("Queue Size: " + queue.size());
        System.out.println("Queue Contains element 'two' or not? : " +
queue.contains("two"));

        // To empty the queue
```

```
    }
}
```

Output:-

```
[one, two, three, four]
[one, two, four]
Queue Size: 3
Queue Contains element 'two' or not? : true
```

Java Array to Queue

Here we can explore how to convert a Java array to Queue using "Collections.addAll()" method with one simple example.

```
import java.util.*;

public class ArrayToQueue {
    public static void main(String[] args) {

        String nums[] = {"one","two","three","four","five"};
        Queue<String> queue = new LinkedList<>();
        Collections.addAll(queue, nums);
        System.out.println(queue);
    }
}
```

Output:-

When we run above program, We will get the following output:

```
[one, two, three, four, five]
```

Java Queue to Array

Here we will explore how to convert a Java Queue to a Java Array using “toArray()” with one simple example.

```
import java.util.*;

public class QueueToArray {
   public static void main(String[] args) {

        Queue<String> queue = new LinkedList<>();
        queue.add("one");
        queue.add("two");
        queue.add("three");
        queue.add("four");
        queue.add("five");

        String strArray[] = queue.toArray(new String[queue.size()]);
        System.out.println(Arrays.toString(strArray));

   }
}
```

Output:-

```
[one, two, three, four, five]
```

Java Queue Common Operations

Java Queue supports all operations supported by Collection interface and some more operations. It supports almost all operations in two forms.

- One set of operations throws an exception if the operation fails.
- The other set of operations returns a special value if the operation fails.

The following table explains all Queue common operations briefly.

OPERATION	THROWS EXCEPTION	SPECIAL VALUE
Insert	add(e)	offer(e)
Remove	remove()	poll()
Examine	element()	peek()

We will pickup each operation and discuss them in-detail with some useful examples in the coming sections.

Java Queue Insert Operations

In this section, we will discuss about Java Queue Insert operation in-detail with some useful examples. If this

forms:

- Queue.add(e):
  It throws an exception if the operation fails.

- Queue.offer(e):
  It returns a special value if the operation fails.

NOTE:- Here special value may be either “false” or “null”

Queue add() operation

The add() operation is used to insert new element into the queue. If it performs insert operation successfully, it returns “true” value. Otherwise it throws java.lang.IllegalStateException.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.concurrent.*;

public class QueueAddOperation {
   public static void main(String[] args) {

        BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);

        System.out.println(queue.add("one"));
        System.out.println(queue.add("two"));
        System.out.println(queue);
        System.out.println(queue.add("three"));
```

```
    }
}
```

Output:-

When we run above program, We will get the following output:

```
true
true
[one, two]
Exception in thread "main" java.lang.IllegalStateException: Queue full
```

As our queue is limited to two elements, when we try to add third element using BlockingQueue.add(), it throws an exception as shown above.

Queue offer() operation

The offer() operation is used to insert new element into the queue. If it performs insert operation successfully, it returns “true” value. Otherwise it returns “false” value.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.concurrent.*;

public class QueueOfferOperation {
   public static void main(String[] args) {

        BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);
```

```
        System.out.println(queue.offer("one"));
        System.out.println(queue.offer("two"));
        System.out.println(queue);
        System.out.println(queue.offer("three"));
        System.out.println(queue);
    }
}
```

Output:-

When we run above program, We will get the following output:

```
true
true
[one, two]
false
[one, two]
```

As our queue is limited to two elements, when we try to add third element using BlockingQueue.offer() operation, it returns "false" value as shown above.

Java Queue Delete Operations

In this section, we will discuss about Java Queue Delete operation in-detail with some useful examples. The Delete operations returns the head element of the queue, if it performs successfully. As we know, Queue supports delete operation in two forms:

- Queue.remove():
  It throws an exception if the operation fails

- Queue.poll():
  It returns a special value if the operation fails.

NOTE:- Here special value may be either “false” or “null”

Queue remove() operation

The remove() operation is used to delete an element from the head of the queue. If it performs delete operation successfully, it returns the head element of the queue. Otherwise it throws java.util.NoSuchElementException.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;

public class QueueRemoveOperation
{
   public static void main(String[] args)
   {
        Queue<String> queue = new LinkedList<>();
        queue.offer("one");
        queue.offer("two");
        System.out.println(queue);
        System.out.println(queue.remove());
        System.out.println(queue.remove());
        System.out.println(queue.remove());
   }
}
```

Output:-

When we run above program, We will get the following output:

```
[one, two]
one
two
Exception in thread "main" java.util.NoSuchElementException
```

As our queue has only two elements, when we try to call remove() method for third time, it throws an exception as shown above.

NOTE:-

Queue.remove(element) is used to delete a specified element from the queue. If it performs delete operation successfully, it returns “true” value. Otherwise it returns “false” value.

Queue poll() operation

The poll() operation is used to delete an element from the head of the queue. If it performs delete operation successfully, it returns the head element of the queue. Otherwise it returns “null” value.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;

public class QueuePollOperation
{
   public static void main(String[] args)
```

```
        Queue<String> queue = new LinkedList<>();
        queue.offer("one");
        queue.offer("two");
        System.out.println(queue);
        System.out.println(queue.poll());
        System.out.println(queue.poll());
        System.out.println(queue.poll());
    }
}
```

Output:-

When we run above program, We will get the following output:

```
[one, two]
one
two
null
```

As our queue has only two elements, when we try to call poll() method for third time, it returns null value as shown above.

Java Queue Examine Operations

In this section, we will discuss about Java Queue Examine operations in-detail with some useful examples. If this operation performs successfully, it returns the head element of the queue without removing it. As we know, Queue supports examine operation in two forms:

- Queue.element():

- Queue.peek():
  It returns a special value if the operation fails.

NOTE:- Here special value may be either “false” or “null”

Queue element() operation

The element() operation is used to retrieve an element from the head of the queue, without removing it. If it performs examine operation successfully, it returns the head element of the queue. Otherwise it throws java.util.NoSuchElementException.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;

public class QueueElementOperation {
   public static void main(String[] args) {

         Queue<String> queue = new LinkedList<>();
         queue.add("one");

         System.out.println(queue.element());
         System.out.println(queue);
         queue.clear();
         System.out.println(queue.element());
   }
}
```

Output:-

When we run above program, We will get the following output:

```
one
[one]
Exception in thread "main" java.util.NoSuchElementException
```

If we try to call element() method on empty Queue, it throws an exception as shown above.

Queue peek() operation

The peek() operation is used to retrieve an element from the head of the queue, without removing it. If it performs examine operation successfully, it returns the head element of the queue. Otherwise it returns null value.

Let us develop one simple example to demonstrate this functionality.

```
import java.util.*;

public class QueuePeekOperation {
   public static void main(String[] args) {

        Queue<String> queue = new LinkedList<>();
        queue.add("one");

        System.out.println(queue.peek());
        System.out.println(queue);
```

```
    }
}
```

Output:-
When we run above program, We will get the following output:

```
one
[one]
null
```

If we try to call peek() method on empty Queue, it returns null value, but does NOT throw an exception as shown above.

Java Queue Categories

In Java, we can find many Queue implementations. W can broadly categorize them into the following two types

- Bounded Queues
- Unbounded Queues

Bounded Queues are queues which are bounded by capacity that means we need to provide the max size of the queue at the time of creation. For example ArrayBlockingQueue (see previous example).

Unbounded Queues are queues which are NOT bounded by capacity that means we should not provide the size of the queue. For example LinkedList (see previous example).

All Queues which are available in java.util package are Unbounded Queues and Queues which are available in java.util.concurrent package are Bounded Queues.

In other ways, W can broadly categorize them into the following two types:

- Blocking Queues
- Non-Blocking Queues

All Queues which implement BlockingQueue interface are BlockingQueues and rest are Non-Blocking Queues.

BlockingQueues blocks until it finishes it's job or time out, but Non-BlockingQueues do not.

Some Queues are Deques and some queues are PriorityQueues.

BlockingQueue Operations

In addition to Queue's two forms of operations, BlockingQueue's supports two more forms as shown below.

OPERATION	THROWS EXCEPTION	SPECIAL VALUE	BLOCKS	TIMES OUT
Insert	add(e)	offer(e)	put(e)	offer(e, time, unit)
Remove	remove()	poll()	take()	poll(time, unit)
Examine	element()	peek()	N/A	N/A

Some operations are blocked until it finishes it's job and other are blocked until time out.

That's all of a quick roundup on Queue in Java. I hope these Java Queue examples will help you in getting started with Queue collection programming.

Please drop me a comment if you like my tutorials or have any suggestions or issues or type errors.

Thank you.

FILED UNDER: JAVA

About Rambabu Posa
Rambabu Posa have 13 years of RICH experience as Sr Agile Lead Java/Scala/BigData/NoSQL Developer. Apart from Java, he is good at Spring4, Hibernate4, MEAN Stack, RESTful WebServices, NoSQL, BigData Hadoop Stack, Cloud, Scala, Groovy Grails, Play Framework, Lagom Framework and ConductR, TDD, BDD,Agile,DevOps and much more. His hobbies are Developing software, Learning new technologies, Love Walking, Reading Books, Watching TV and obviously sharing his knowledge through writing articles on JournalDev.

« Best Online Programming Courses
Java Set – Set in Java »

Comments

kevin hu says

Good tutorials, thanks so much!

Reply

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT