PriorityBlockingQueue Class in Java

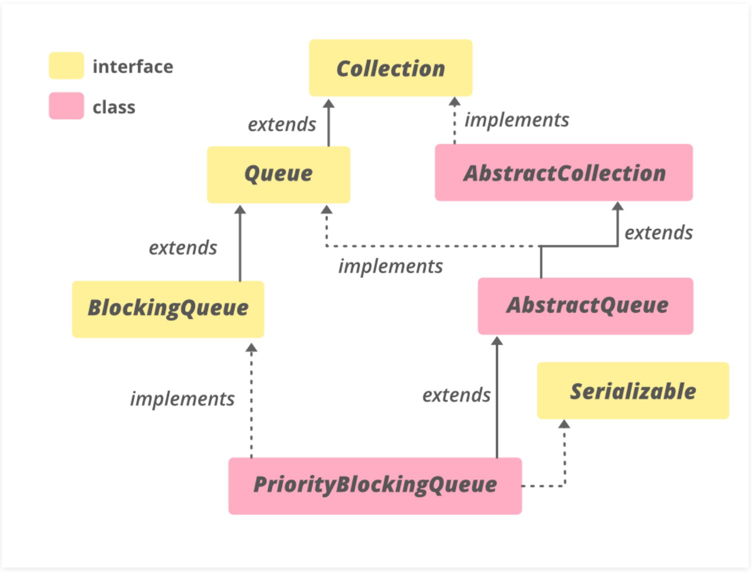
Last Updated : 06 Nov, 2020

The **PriorityBlockingQueue** is an unbounded blocking queue that uses the same ordering rules as class **PriorityQueue** and supplies blocking retrieval operations. Since it is unbounded, adding elements may sometimes fail due to resource exhaustion resulting in **OutOfMemoryError**. This class does not permit null elements.
PriorityBlockingQueue class and its iterator implement all of the optional methods of the Collection and Iterator interfaces.

The Iterator provided in method iterator() and the Spliterator provided in the method spliterator() are not guaranteed to traverse the elements of the PriorityBlockingQueue in any particular order. For ordered traversal, use **Arrays.sort(pq.toArray())**. Also, method drainTo() can be used to remove some or all elements in priority order and place them in another collection.

Operations on this class make no guarantees about the ordering of elements with equal priority. If an ordering is needed to be enforced, define custom classes or comparators that use a secondary key to break ties in primary priority values.
This class is a member of the Java Collections Framework.

The Hierarchy of PriorityBlockingQueue



It
implements **Serializable**, **Iterable<E>**, **Collection<E>**, BlockingQueue<E>, Queue<E> interfaces and extends AbstractQueue<E> class.

Declaration:

public class PriorityBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable

Here, **E** is the type of elements held in this collection.

Constructors of PriorityBlockingQueue

In order to create an instance of PriorityBlockingQueue, we need to import it from **java.util.concurrent.PriorityBlockingQueue**.

1. PriorityBlockingQueue() – Creates a PriorityBlockingQueue with the default initial capacity (11) that orders its elements according to their natural ordering. Adding element more than the initial capacity changes the capacity of the PriorityBlockingQueue dynamically as the PriorityBlockingQueue is not capacity constrained.

PriorityBlockingQueue<E> pbq = new PriorityBlockingQueue<E>();

Example:

```java
// Java program to demonstrate
// PriorityBlockingQueue() constructor

import java.util.concurrent.PriorityBlockingQueue;

public class GFG {

    public static void main(String[] args)
    {

        // create object of PriorityBlockingQueue
        // using PriorityBlockingQueue() constructor
        PriorityBlockingQueue<Integer> pbq
            = new PriorityBlockingQueue<Integer>();

        // add  numbers
        pbq.add(1);
        pbq.add(2);
        pbq.add(3);
        pbq.add(4);
        pbq.add(5);
        // print queue
        System.out.println("PriorityBlockingQueue:" + pbq);
    }
}
```

Output:

```
PriorityBlockingQueue:[1, 2, 3, 4, 5]
```

2. PriorityBlockingQueue(Collection<? extends E> c) – Creates a PriorityBlockingQueue containing the elements in the specified collection.

PriorityBlockingQueue<E> pbq = new PriorityBlockingQueue(Collection<? extends E> c);

Example:

```
// Java program to demonstrate
// PriorityBlockingQueue(Collection c) constructor

import java.util.concurrent.PriorityBlockingQueue;
import java.util.*;

public class GFG {

    public static void main(String[] args)
    {

        // Creating a Collection
        Vector<Integer> v = new Vector<Integer>();
        v.addElement(1);
        v.addElement(2);
        v.addElement(3);
        v.addElement(4);
        v.addElement(5);

        // create object of PriorityBlockingQueue
        // using PriorityBlockingQueue(Collection c)
        // constructor
        PriorityBlockingQueue<Integer> pbq
```

```
                = new PriorityBlockingQueue<Integer>(v);

            // print queue
            System.out.println("PriorityBlockingQueue:" + pbq);
        }
    }
```

Output:

```
PriorityBlockingQueue:[1, 2, 3, 4, 5]
```

3. PriorityBlockingQueue(int initialCapacity) – Creates a PriorityBlockingQueue with the specified initial capacity that orders its elements according to their natural ordering.

Example:

```
// Java program to demonstrate
// PriorityBlockingQueue(int initialCapacity)
// constructor

import java.util.concurrent.PriorityBlockingQueue;

public class GFG {

    public static void main(String[] args)
    {
        // define capacity of PriorityBlockingQueue
        int capacity = 15;

        // create object of PriorityBlockingQueue
        // using PriorityBlockingQueue(int initialCapacity)
        // constructor
        PriorityBlockingQueue<Integer> pbq
            = new PriorityBlockingQueue<Integer>(capacity);

        // add  numbers
        pbq.add(1);
        pbq.add(2);
        pbq.add(3);

        // print queue
        System.out.println("PriorityBlockingQueue:" + pbq);
    }
}
```

Output:

```
PriorityBlockingQueue:[1, 2, 3]
```

4. PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator) – Creates a PriorityBlockingQueue with the specified initial capacity that orders its elements according to the specified comparator.

Example:

```
// Java program to demonstrate
// PriorityBlockingQueue(int initialCapacity, Comparator
// comparator) constructor

import java.util.concurrent.PriorityBlockingQueue;
import java.util.*;

public class GFG {

    public static void main(String[] args)
    {
        // define capacity of PriorityBlockingQueue
        int capacity = 15;

        // create object of PriorityBlockingQueue
        PriorityBlockingQueue<Integer> pbq
            = new PriorityBlockingQueue<Integer>(
                capacity, Comparator.reverseOrder());

        // add  numbers
        pbq.add(1);
        pbq.add(2);
        pbq.add(3);

        // print queue
        System.out.println("PriorityBlockingQueue:" + pbq);
    }
}
```

Output:

```
PriorityBlockingQueue:[3, 1, 2]
```

Basic Operations

1. Adding Elements

The add(E e) method of PriorityBlockingQueue inserts the element passed as a parameter to the method at the tail of this PriorityBlockingQueue. This method returns true if the adding of the element is successful. Else it returns false.

```java
// Java program to demonstrate adding elements
// to the PriorityBlockingQueue

import java.util.concurrent.PriorityBlockingQueue;

public class AddingElementsExample {

    public static void main(String[] args)
    {
        // define capacity of PriorityBlockingQueue
        int capacity = 15;

        // create object of PriorityBlockingQueue
        PriorityBlockingQueue<Integer> pbq
            = new PriorityBlockingQueue<Integer>(capacity);

        // add  numbers
        pbq.add(1);
        pbq.add(2);
        pbq.add(3);

        // print queue
        System.out.println("PriorityBlockingQueue:" + pbq);
    }
}
```

Output:

```
PriorityBlockingQueue:[1, 2, 3]
```

2. Removing Elements

The remove(Object o) method of PriorityBlockingQueue is used to delete an element from this queue. This method removes a single instance of the element passed as the parameter if it is present. It returns true if and only if the element was removed, else it returned false. clear() is used to remove all the elements at once.

```java
// Java program to demonstrate removing
// elements from the PriorityBlockingQueue

import java.util.concurrent.PriorityBlockingQueue;

public class RemovingElementsExample {

    public static void main(String[] args)
    {
        // define capacity of PriorityBlockingQueue
        int capacity = 15;

        // create object of PriorityBlockingQueue
        PriorityBlockingQueue<Integer> pbq
            = new PriorityBlockingQueue<Integer>(capacity);

        // add  numbers
        pbq.add(1);
        pbq.add(2);
        pbq.add(3);

        // print queue
        System.out.println("PriorityBlockingQueue:" + pbq);

        // remove all the elements
        pbq.clear();

        // print queue
        System.out.println("PriorityBlockingQueue:" + pbq);
    }
}
```

Output:

```
PriorityBlockingQueue:[1, 2, 3]
PriorityBlockingQueue:[]
```

3. Accessing Elements

The peek() method of PriorityBlockingQueue returns the element at the head of the PriorityBlockingQueue. It retrieves the value of the head of LinkedBlockingQueue but does not remove it. If the PriorityBlockingQueue does not contain any element, then this method returns null. A PriorityBlockingQueue queue uses the same ordering rules as class PriorityQueue.

```java
// Java Program Demonstrate accessing
// elements of PriorityBlockingQueue

import java.util.concurrent.PriorityBlockingQueue;

public class AccessingElementsExample {
    public static void main(String[] args)
    {
        // define capacity of PriorityBlockingQueue
        int capacityOfQueue = 5;

        // create object of PriorityBlockingQueue
        PriorityBlockingQueue<Integer> PrioQueue
            = new PriorityBlockingQueue<Integer>(
                capacityOfQueue);

        // Add elements to PriorityBlockingQueue
        PrioQueue.add(464161);
        PrioQueue.add(416165);

        // print PrioQueue
        System.out.println("PrioQueue: " + PrioQueue);

        // get head of PriorityBlockingQueue
        int head = PrioQueue.peek();

        // print head of PriorityBlockingQueue
        System.out.println("Head of Queue: " + head);
    }
}
```

Output

```
PrioQueue: [416165, 464161]
Head of Queue: 416165
```

4. Iterating

The iterator() method of PriorityBlockingQueue class Returns an iterator over the elements in this queue. The elements returned from this method do not follow any order. The returned iterator is weakly consistent.

```java
// Java Program Demonstrate iterating
// over PriorityBlockingQueue

import java.util.concurrent.PriorityBlockingQueue;
import java.util.*;

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

        // define capacity of PriorityBlockingQueue
        int capacityOfQueue = 5;

        // create object of PriorityBlockingQueue
        PriorityBlockingQueue<String> names
            = new PriorityBlockingQueue<String>(
                capacityOfQueue);

        // Add names of students of girls college
        names.add("Geeks");
        names.add("forGeeks");
        names.add("A");
        names.add("Computer");
        names.add("Portal");

        // Call iterator() method of PriorityBlockingQueue
        Iterator iteratorVals = names.iterator();

        // Print elements of iterator
        // created from PriorityBlockingQueue
        System.out.println("The Names are:");

        while (iteratorVals.hasNext()) {
            System.out.println(iteratorVals.next());
        }
    }
}
```

Output

```
The Names are:
A
Computer
Geeks
forGeeks
Portal
```

5. Comparator Example

The comparator() method of PriorityBlockingQueue returns the comparator that can be used to order the elements in a PriorityBlockingQueue. The method returns a null value if the queue follows the natural ordering pattern of the elements.

```java
// Java Program Demonstrate comparator()
// method and passing Comparator to PriorityBlockingQueue

import java.util.concurrent.PriorityBlockingQueue;
import java.util.*;

public class ComparatorExample {
    public static void main(String[] args)
        throws InterruptedException
    {

        // create object of PriorityBlockingQueue
        PriorityBlockingQueue<Integer> PrioQueue
            = new PriorityBlockingQueue<Integer>(
                10, new Comparator<Integer>() {
                    public int compare(Integer a, Integer b)
                    {
                        return a - b;
                    }
                });

        // Add numbers to PriorityBlockingQueue
        PrioQueue.put(45815616);
        PrioQueue.put(4981561);
        PrioQueue.put(4594591);
        PrioQueue.put(9459156);

        // get String representation of
        // PriorityBlockingQueue
        String str = PrioQueue.toString();

        // Creating a comparator using comparator()
        Comparator comp = PrioQueue.comparator();
        // Displaying the comparator values
        System.out.println("Comparator value: " + comp);

        if (comp == null)
            System.out.println(
                "PriorityBlockingQueue follows natural ordering");
        else
            System.out.println(
                "PriorityBlockingQueue follows : " + comp);
    }
}
```

Output

```
Comparator value: ComparatorExample$1@27bc2616
PriorityBlockingQueue follows : ComparatorExample$1@27bc2616
```

Methods of PriorityBlockingQueue

METHOD	DESCRIPTION
add(E e)	Inserts the specified element into this priority queue.
clear()	Atomically removes all of the elements from this queue.
comparator()	Returns the comparator used to order the elements in this queue, or null if this queue uses the natural ordering of its elements.
contains(Object o)	Returns true if this queue contains the specified element.
drainTo(Collection<? super E> c)	Removes all available elements from this queue and adds them to the given collection.
drainTo(Collection<? super E> c, int maxElements)	Removes at most the given number of available elements from this queue and adds them to the given collection.
forEach(Consumer<? super E> action)	Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
iterator()	Returns an iterator over the elements in this queue.
offer(E e)	Inserts the specified element into this priority queue.

METHOD	DESCRIPTION
offer(E e, long timeout, TimeUnit unit)	Inserts the specified element into this priority queue.
put(E e)	Inserts the specified element into this priority queue.
remainingCapacity()	Always returns Integer.MAX_VALUE because a PriorityBlockingQueue is not capacity constrained.
remove(Object o)	Removes a single instance of the specified element from this queue, if it is present.
removeAll(Collection<?> c)	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
retainAll(Collection<?> c)	Retains only the elements in this collection that are contained in the specified collection (optional operation).
spliterator()	Returns a Spliterator over the elements in this queue.
toArray()	Returns an array containing all of the elements in this queue.
toArray(T[] a)	Returns an array containing all of the elements in this queue; the runtime type of the returned array is that of the specified array.

Methods declared in class java.util.AbstractQueue

METHOD	DESCRIPTION
addAll(Collection<? extends E> c)	Adds all of the elements in the specified collection to this queue.
element()	Retrieves, but does not remove, the head of this queue.
remove()	Retrieves and removes the head of this queue.

Methods declared in class java.util.AbstractCollection

METHOD	DESCRIPTION
containsAll (Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
isEmpty()	Returns true if this collection contains no elements.
toString()	Returns a string representation of this collection.

Methods declared in interface java.util.concurrent.BlockingQueue

METHOD	DESCRIPTION
poll(long timeout, TimeUnit unit)	Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
take()	Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Methods declared in interface java.util.Collection

METHOD	DESCRIPTION
addAll (Collection<? extends E> c)	Adds all of the elements in the specified collection to this collection (optional operation).
containsAll (Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
equals(Object o)	Compares the specified object with this collection for equality.
hashCode()	Returns the hash code value for this collection.
isEmpty()	Returns true if this collection contains no elements.
parallelStream()	Returns a possibly parallel Stream with this collection as its source.
size()	Returns the number of elements in this collection.
stream()	Returns a sequential Stream with this collection as its source.

METHOD	DESCRIPTION
toArray (IntFunction<T[]> generator)	Returns an array containing all of the elements in this collection, using the provided generator function to allocate the returned array.

Methods declared in interface java.util.Queue

METHOD	DESCRIPTION
element()	Retrieves, but does not remove, the head of this queue.
peek()	Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
poll()	Retrieves and removes the head of this queue, or returns null if this queue is empty.
remove()	Retrieves and removes the head of this queue.

Reference: https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/PriorityBlockingQueue.html