CSCI 2120:
Software Design & Development II

UNIT2: Data management

Collections Framework & Generics

Deque

Overview

1. Introduction
2. Hierarchy of Deque interface
3. Deque interface declaration
4. Features of Deque
5. Deque Methods
6. Deque Examples

Introduction

A **double ended queue** or **deque in Java** is a linear data structure that supports insertion and removal of elements from both ends (head and tail). It is usually pronounced “deck”, not “de queue”.

Java deque is an extended version of queue to handle a double-ended queue. It extends Queue interface with additional methods for adding and removing elements from both ends of queues.

Deque interface was recently introduced in Java 1.6 version. It is now part of the Java collections framework. It is present in java.util.Deque package.

Double ended queue in Java can be used as FIFO (first-in, first-out) queue or LIFO (last-in, first-out) queue. Adding elements in the middle of queue is not supported.

Hierarchy of Java Deque interface

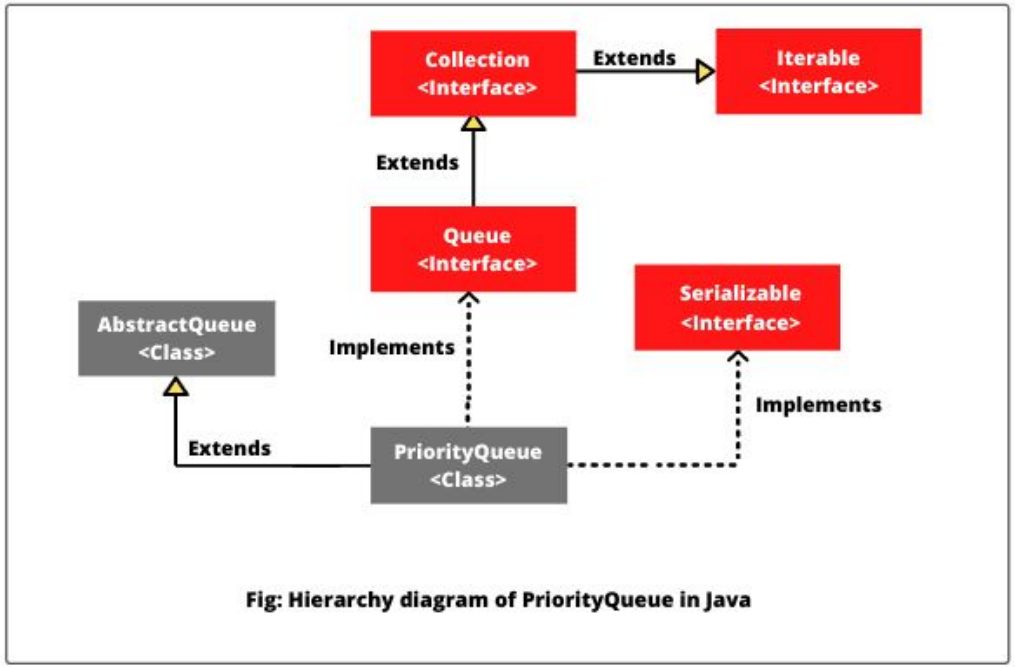

Fig: Hierarchy diagram of PriorityQueue in Java

Deque interface extends Queue interface to handle double ended queue. It is implemented by LinkedList and ArrayDeque classes, both of which can be used to create a queue whose size can grow as needed.

LinkedList is ideal for queue operations because it is efficient for adding and removing elements from both ends of a list.

Other implementation classes are ConcurrentLinkedDeque, and LinkedBlockingDeque which also implement the deque interface.

Deque interface declaration

Deque is a generic interface that can be declared as follows:

```
public interface Deque<E> extends Queue<E>
```

Here, E represents the type of objects that deque will hold.

Features of Double ended queue

There are several interesting features of the deque in java that is as follows:

1. Java Deque interface orders elements in First In First Out or Last In First Out policy.

2. Deque does not allow to store null elements. If we try to add, `NullPointerException` will be thrown.

3. The most important features of Deque are push and pop. The push() and pop() methods are commonly used to enable a Deque to function as a stack.

To put an element on the top of the stack, call `push()` method. To remove the top element, call `pop()` method.

Features of Double ended queue

There are several interesting features of the deque in java that is as follows:

4. Elements can be retrieved and removed from both ends of the queue.

5. Elements can be added from both ends of the queue.

6. The `LinkedList` and `ArrayDeque` classes are two implementation classes for the `Deque` interface.

7. `ArrayDeque` class can be implemented if deque is used as a LIFO queue (or a stack).

8. The `LinkedList` implementation performs better if deque is used as a FIFO queue (or simply as a queue).

Deque Methods

In addition to methods inheriting from the Collection interface and Queue, Deque interface adds 17 new methods to facilitate insertion, removal, and peeking operations at both ends. They are as follows:

Deque Methods

Method	Description
void addFirst(E e)	This method is used to add an element at the head of the deque. It throws an exception named IllegalStateException if a capacity-restricted deque is out of space.
void addLast(E e)	This method is used to add an element at the tail of the deque. It throws an IllegalStateException if a capacity-restricted deque is out of space.
boolean offerFirst(E e)	It is used to add an element at the head of the deque. It returns true if the element is added successfully in the deque otherwise returns false. However, it does not throw an exception on failure.
boolean offerLast(E e)	It is used to add an element at the tail of the deque. It returns true if the element is added successfully in the deque otherwise returns false. However, it does not throw an exception on failure.
E removeFirst( )	This method is used to retrieve and remove the element from the head of the deque. They throw an exception named NoSuchElementException if the deque is empty.

Deque Methods

Method	Description
E removeLast()	This method is used to retrieve and remove the element from the tail of the deque. They throw an exception named NoSuchElementException if the deque is empty.
E pollFirst( )	The pollFirst() method performs the same task as the removeFirst() method. However, it returns null if the deque is empty.
E pollLast()	The pollLast() method performs the same task as the removeLast() method. However, it returns null if the deque is empty.
E getFirst( )	It is used to retrieve without removing the first element from the head of deque. It throws NoSuchElementException if the deque is empty.
E getLast()	It is used to retrieve without removing the last element from the tail of deque. It throws NoSuchElementException if the deque is empty.

Deque Methods

Method	Description
E peekFirst( )	The peekFirst() method works the same task as the getFirst() method. It returns null object if deque is empty instead of throwing an exception.
E peekLast()	The peekLast() method works the same task as the getLast() method. It returns null object if deque is empty instead of throwing an exception.
void push(E e)	The push() method adds (or pushes) an element to the head of the deque. It will throw an IllegalStateException if a capacity-restricted deque is out of space. This method is the same as addFirst() method.
E pop( )	The pop() method removes (or pops) an element from the head of deque. If the deque is empty, it will throw NoSuchElementException. This method is the same as removeFirst() method.

Deque Methods

Method	Description
boolean removeFirstOccurrence(Object o)	This method removes the first occurrence of the specified element from the deque. It returns true if successful removed and false if the deque did not contain the specified element.
boolean removeLastOccurrence(Object o)	This method removes the last occurrence of the specified element from the deque. It will return true if successful removed and false if the deque did not contain the specified element.
Iterator<E> descendingIterator( )	It returns an iterator object that iterate over its elements in reverse order (from tail to head). The descendingIterator() method works the same as the iterator() method, except that it moves in the opposite direction.

Deque Example

Example 1: gets, removes, adds

Let’s take an example program where we will perform operations based on getFirst(), removeFirst(), addFirst(), getLast(), removeLast(), and addLast() methods. We will implement ArrayDeque class for deque interface.

Example 1: gets, removes, adds

```
import java.util.ArrayDeque;
import java.util.Deque;
public class DequeTester1 {
   public static void main(String[] args) {
       Deque<String> dq = new ArrayDeque<String>();    // Create a Deque and add elements to the deque.
       dq.offer("ABC");
       dq.offer("PQR");
       dq.offer("MNO");
       dq.offer("IJK");
       dq.offer("GHI");
       System.out.println(dq);
       System.out.println("dq.getFirst(): " +dq.getFirst());
       System.out.println(dq);
       System.out.println("dq.removeFirst(): " +dq.removeFirst());
       System.out.println(dq);

       dq.addFirst("ABC");                          // Adding new element at the head of queue.
       System.out.println(dq);

       System.out.println("dq.getLast(): " +dq.getLast());
       System.out.println(dq);
       System.out.println("dq.removeLast(): " +dq.removeLast());
       System.out.println(dq);

       dq.addLast("GHI");
       System.out.println(dq);
   }
}
```

Example 1: gets, removes, adds

Output:

```
[ABC, PQR, MNO, IJK, GHI]
dq.getFirst(): ABC
[ABC, PQR, MNO, IJK, GHI]
dq.removeFirst(): ABC
[PQR, MNO, IJK, GHI]
[ABC, PQR, MNO, IJK, GHI]
dq.getLast(): GHI
[ABC, PQR, MNO, IJK, GHI]
dq.removeLast(): GHI
[ABC, PQR, MNO, IJK]
[ABC, PQR, MNO, IJK, GHI]
```

Example 2: LinkedList as Deque as a FIFO queue

Let’s create a program where we will use a deque as a FIFO queue (or simply as a queue). We will implement LinkedList class for the deque interface. LinkedList class performs better if we use deque as a FIFO queue. Look at the source code to understand better.

Example 2: LinkedList as Deque as a FIFO queue

```
import java.util.Deque;
import java.util.LinkedList;
import java.util.NoSuchElementException;
public class DequeTester2 {
   public static void main(String[] args) {
       Deque<String> dq = new LinkedList<>();                    // Create Deque, add elements to tail with addLast() or offerLast()
       dq.addLast("John");
       dq.offerLast("Richard");
       dq.offerLast("Donna");
       dq.offerLast("Ken");
       dq.offer("Peter");
       System.out.println("Deque: " + dq);
       while (dq.peekFirst() != null) {                          // Remove elements from deque until it is empty.
           System.out.println("Head Element: " + dq.peekFirst());
           System.out.println("Removed element from Deque: " +dq.removeFirst());
           System.out.println("Elements in Deque: " + dq);
       }
       System.out.println("\n");                                 // Now, deque is empty. Try peekFirst, getFirst, pollFirst, removeFirs.
       System.out.println("deque.isEmpty(): " + dq.isEmpty());
       System.out.println("deque.peekFirst(): " + dq.peekFirst());
       System.out.println("deque.pollFirst(): " + dq.pollFirst());
       try {
           String str = dq.getFirst();
           System.out.println("deque.getFirst(): " + str);
       } catch (NoSuchElementException e) {
           System.out.println("deque.getFirst(): Deque is empty.");
       }
       try {
           String str = dq.removeFirst();
           System.out.println("deque.removeFirst(): " + str);
       } catch (NoSuchElementException e) {
           System.out.println("deque.removeFirst(): Deque is empty.");
       }
   }
}
```

Example 2: LinkedList as Deque as a FIFO queue

Output:

```
Deque: [John, Richard, Donna, Ken, Peter]
Head Element: John
Removed element from Deque: John
Elements in Deque: [Richard, Donna, Ken, Peter]
Head Element: Richard
Removed element from Deque: Richard
Elements in Deque: [Donna, Ken, Peter]
Head Element: Donna
Removed element from Deque: Donna
Elements in Deque: [Ken, Peter]
Head Element: Ken
Removed element from Deque: Ken
Elements in Deque: [Peter]
Head Element: Peter
Removed element from Deque: Peter
Elements in Deque: []


deque.isEmpty(): true
deque.peekFirst(): null
deque.pollFirst(): null
deque.getFirst(): Deque is empty.
deque.removeFirst(): Deque is empty.
```

Example 3: ArrayDeque as Deque as a LIFO queue

Let’s create another program where we will demonstrate how to use a Deque as a stack (or LIFO queue). We will implement ArrayDeque class for deque interface.

Example 3: ArrayDeque as Deque as a LIFO queue

```
import java.util.ArrayDeque;
import java.util.Deque;
public class DequeTester3{
   public static void main(String[] args) {
       // Create a Deque and use it as stack. add elements at its tail using addLast() or offerLast()
method.
       Deque<String> dq = new ArrayDeque<>();

       dq.push("John");
       dq.push("Richard");
       dq.push("Donna");
       dq.push("Ken");
       dq.push("Peter");

       System.out.println("Stack: " + dq);

       // Remove all elements from the deque.
       while (dq.peek() != null) {
           System.out.println("Element at top: " + dq.peek());
           System.out.println("Popped: " + dq.pop());
           System.out.println("Stack: " + dq);
       }
       System.out.println(" Is Stack empty: " + dq.isEmpty());
   }
}
```

Example 3: ArrayDeque as Deque as a LIFO queue

Output:

```
Stack: [Peter, Ken, Donna, Richard, John]
Element at top: Peter
Popped: Peter
Stack: [Ken, Donna, Richard, John]
Element at top: Ken
Popped: Ken
Stack: [Donna, Richard, John]
Element at top: Donna
Popped: Donna
Stack: [Richard, John]
Element at top: Richard
Popped: Richard
Stack: [John]
Element at top: John
Popped: John
Stack: []
 Is Stack empty: true
```

Example 4: Iterate over Deque with iterator

Let’s take an example program where we will iterate over elements of deque using iterator() method. The iterator() method returns an iterator object that iterates over elements of a deque in LIFO order. i.e. the elements will be traversed from head to tail.

Example 4: Iterate over Deque with iterator

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Iterator;
public class DequeTester4 {
   public static void main(String[] args) {
       Deque<Integer> dq = new ArrayDeque<Integer>();
       dq.offer(50);
       dq.offer(10);
       dq.offer(20);
       dq.offer(05);
       dq.offer(30);
       System.out.println("Elements in deque:");
       System.out.println(dq);

       // Iterating over elements of the deque.
       System.out.println("\nIteration in forward direction:");
       Iterator<Integer> itr = dq.iterator();
       while(itr.hasNext()) {
           System.out.println(itr.next());
       }
       // Iterating over elements in reverse order.
       System.out.println("\nIteration in reverse order:");
       Iterator<Integer> itr2 = dq.descendingIterator();
       while(itr2.hasNext()) {
           System.out.println(itr2.next());
       }
   }
}
```

Example 4: Iterate over Deque with iterator

Output:

```
Elements in deque:
[50, 10, 20, 5, 30]

Iteration in forward direction:
50
10
20
5
30

Iteration in reverse order:
30
5
20
10
50
```

Example 5: Iterate over Deque with enhanced for-loop

Let’s take another example program where we will use enhanced for loop to iterate over elements of deque in LIFO order. Look at the source code.

Example 5: Iterate over Deque with enhanced for-loop

```
import java.util.ArrayDeque;
import java.util.Deque;
public class DequeTester5 {
   public static void main(String[] args) {
       Deque<Integer> dq = new ArrayDeque<Integer>();

       dq.offer(50);
       dq.offer(10);
       dq.offer(20);
       dq.offer(05);
       dq.offer(30);

       // Iterating over elements of deque using enhanced for loop.
       System.out.println("Iterating using enhanced for loop");
       for (Integer element: dq) {
           System.out.println(element);
       }
   }
}
```

Example 5: Iterate over Deque with enhanced for-loop

Output:

```
Iterating using enhanced for loop
50
10
20
5
30
```

END