**Java Collection - 2022**

**Java Collection Highlights**

|  |  |  |
| --- | --- | --- |
| **Type** | **Normal** | **Concurrent** |
| Deque | [LinkedList](file:///E:\dev\java7\docs\api\java\util\LinkedList.html), [ArrayDeque](file:///E:\dev\java7\docs\api\java\util\ArrayDeque.html) | [LinkedBlockingDeque](file:///E:\dev\java7\docs\api\java\util\concurrent\LinkedBlockingDeque.html), [ConcurrentLinkedDeque](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentLinkedDeque.html), |
| List | [ArrayList](file:///E:\dev\java7\docs\api\java\util\ArrayList.html) , [LinkedList](file:///E:\dev\java7\docs\api\java\util\LinkedList.html) , [Stack](file:///E:\dev\java7\docs\api\java\util\Stack.html), [Vector](file:///E:\dev\java7\docs\api\java\util\Vector.html) | [CopyOnWriteArrayList](file:///E:\dev\java7\docs\api\java\util\concurrent\CopyOnWriteArrayList.html) |
| Map | [HashMap](file:///E:\dev\java7\docs\api\java\util\HashMap.html), [LinkedHashMap](file:///E:\dev\java7\docs\api\java\util\LinkedHashMap.html), [TreeMap](file:///E:\dev\java7\docs\api\java\util\TreeMap.html)  [Hashtable](file:///E:\dev\java7\docs\api\java\util\Hashtable.html), [EnumMap](file:///E:\dev\java7\docs\api\java\util\EnumMap.html), [WeakHashMap](file:///E:\dev\java7\docs\api\java\util\WeakHashMap.html)  [IdentityHashMap](file:///E:\dev\java7\docs\api\java\util\IdentityHashMap.html), [Properties](file:///E:\dev\java7\docs\api\java\util\Properties.html) | [ConcurrentHashMap](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentHashMap.html), [ConcurrentSkipListMap](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListMap.html) |
| NavigableMap | [TreeMap](file:///E:\dev\java7\docs\api\java\util\TreeMap.html) | [ConcurrentSkipListMap](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListMap.html) |
| NavigableSet | [TreeSet](file:///E:\dev\java7\docs\api\java\util\TreeSet.html) | [ConcurrentSkipListSet](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListSet.html) |
| Queue | [PriorityQueue](file:///E:\dev\java7\docs\api\java\util\PriorityQueue.html), [ArrayDeque](file:///E:\dev\java7\docs\api\java\util\ArrayDeque.html) , [LinkedList](file:///E:\dev\java7\docs\api\java\util\LinkedList.html) | [**ArrayBlockingQueue**](file:///E:\dev\java7\docs\api\java\util\concurrent\ArrayBlockingQueue.html), [**PriorityBlockingQueue**](file:///E:\dev\java7\docs\api\java\util\concurrent\PriorityBlockingQueue.html),  [**LinkedBlockingQueue**](file:///E:\dev\java7\docs\api\java\util\concurrent\LinkedBlockingQueue.html), [LinkedBlockingDeque](file:///E:\dev\java7\docs\api\java\util\concurrent\LinkedBlockingDeque.html)  [ConcurrentLinkedDeque](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentLinkedDeque.html), [**ConcurrentLinkedQueue**](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentLinkedQueue.html)  [DelayQueue](file:///E:\dev\java7\docs\api\java\util\concurrent\DelayQueue.html), [LinkedTransferQueue](file:///E:\dev\java7\docs\api\java\util\concurrent\LinkedTransferQueue.html) ,  [SynchronousQueue](file:///E:\dev\java7\docs\api\java\util\concurrent\SynchronousQueue.html) |
| RandomAccess | [ArrayList](file:///E:\dev\java7\docs\api\java\util\ArrayList.html), [Stack](file:///E:\dev\java7\docs\api\java\util\Stack.html), [Vector](file:///E:\dev\java7\docs\api\java\util\Vector.html), [AttributeList](file:///E:\dev\java7\docs\api\javax\management\AttributeList.html), [RoleList](file:///E:\dev\java7\docs\api\javax\management\relation\RoleList.html), [RoleUnresolvedList](file:///E:\dev\java7\docs\api\javax\management\relation\RoleUnresolvedList.html), | [CopyOnWriteArrayList](file:///E:\dev\java7\docs\api\java\util\concurrent\CopyOnWriteArrayList.html) |
| Set | [EnumSet](file:///E:\dev\java7\docs\api\java\util\EnumSet.html), [HashSet](file:///E:\dev\java7\docs\api\java\util\HashSet.html) , [LinkedHashSet](file:///E:\dev\java7\docs\api\java\util\LinkedHashSet.html), [TreeSet](file:///E:\dev\java7\docs\api\java\util\TreeSet.html) | [ConcurrentSkipListSet](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListSet.html),  [CopyOnWriteArraySet](file:///E:\dev\java7\docs\api\java\util\concurrent\CopyOnWriteArraySet.html) |
| SortedMap | [TreeMap](file:///E:\dev\java7\docs\api\java\util\TreeMap.html) | [ConcurrentSkipListMap](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListMap.html) |
| SortedSet | [TreeSet](file:///E:\dev\java7\docs\api\java\util\TreeSet.html) | [ConcurrentSkipListSet](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListSet.html) |

**ArrayList**

Initial capacity of an ArrayList is 10. The capacity is calculated like the following formula.

**int newCapacity = (oldCapacity \* 3)/2 + 1 🡺 Deprecated**

**int newCapacity = ArraysSupport.newLength(oldCapacity, minCapacity - oldCapacity, oldCapacity >> 1;**

**int** prefLength = oldLength + Math.*max*(minGrowth, prefGrowth);

**int** oldCapacity = elementData.length;

**int oldCapacity = elementData.length; 🡺 use this for answering in interview**

**int newCapacity = oldCapacity + (oldCapacity >> 1);**

**Important methods of ArrayList**

**retainAll(): This method is used to contain the intersection of the two list.**

**List<String> list1 = new ArrayList<String>(Arrays.asList("A", "B", "C"));**

**List<String> list2 = new ArrayList<String>(Arrays.asList("B", "C", "D", "E", "F"));**

**list1.retainAll(list2);**

System.*out*.println(list1); // [B, C]

System.*out*.println(list2); // [B, C, D, E, F]

list2.retainAll(list1);

System.*out*.println(list1); // [A, B, C]

System.*out*.println(list2); // [B, C]

**What is the difference between poll() and pop() ?**

**While retrieving the object**, **both the method performs the same, it means they obtain the head element and remove the element**. The main difference is **poll() is a method of Queue and pop() is a method of Stack**. Let us see a complete program.

LinkedList<Integer> ll = new LinkedList<>();  
ll.add(1); ll.add(2); ll.add(3); ll.add(4); ll.add(5);

System.*out*.println("Peek: "+ ll.peek()); *// 1*System.*out*.println("Poll: "+ll.poll()); *// 1*System.*out*.println(ll); *// 2, 3, 4, 5*System.*out*.println("Pop: "+ ll.pop()); *// 2*System.*out*.println(ll); *// 3,4,5*

**Stack** Important Methods for **Stack** 🡺 **push()** and **pop() 🡺 Stack is LIFO**

The Stack class represents a **last-in-first-out (LIFO)** stack of objects. It is a legacy class.

It extends class Vector with five operations that allow a vector to be treated as a stack.

A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations.

Stack<Integer> stack = new Stack<>();  
stack.add(1);  
stack.add(2);  
stack.add(3);  
stack.add(4);  
stack.add(5);  
System.*out*.println(stack); // [1, 2, 3, 4, 5]

The output of Stack will be same if you use push() or add() method. The data will be stored in sequential manner.

**System.*out*.println(stack.pop()); // 5**

**Only retrieval is important, in case of pop(), it retrieves the last element only.**

In case of Stack, it has methods like add() and push().

It has also methods like pop() and get().

**Queue** Important Methods for **Queue** 🡺 **offer()** and **poll() 🡺** Queue is **FIFO**.

A collection designed for holding elements prior to processing. LinkedList is a complete implementation of Queue. PriorityQueue is also an implementation for Queue.

Queue<Integer> queue = new LinkedList<>();  
queue.add(1);  
queue.add(2);  
queue.add(3);  
queue.add(4);  
queue.add(5);  
System.*out*.println(queue); *//[1, 2, 3, 4, 5]*  
System.*out*.println(queue.poll()); *// 1*

**It does not matter whether you use add() or offer() method in Queue, the data will be stored in sequential manner.** **Only retrieval is important**. You can use LinkedList or PriorityQueue for a queue implementation. **poll() method always returns the first element in the structure**. Poll() method retrieves the value and then deletes the value.

**Deque**

It is a linear collection that supports **element insertion and removal at both ends**. ArrayDeque() : Constructs an empty array deque with an initial capacity sufficient to hold 16 elements.

**If ArrayDeque used as Stack**

ArrayDeque<Integer> deque = new ArrayDeque<>();

deque.push(1);  
deque.push(2);  
deque.push(3);  
deque.push(4);  
deque.push(5);

System.*out*.println(deque); // 🡺 **[5, 4, 3, 2, 1]**  
System.*out*.println(deque.pop()); // 5

**In case of ArrayDeque as Stack, the data are organized in the reverse order of insertion**. Both pop() and poll() retrieves the first element.

**If ArrayDeque used as Queue**

ArrayDeque<Integer> deque = new ArrayDeque<>();

deque.offer(1);  
deque.offer(2);  
deque.offer(3);  
deque.offer(4);  
deque.offer(5);

System.*out*.println(deque); // 🡺 [1, 2, 3, 4, 5]  
System.*out*.println(deque.poll()); // 1

**In case of ArrayDeque as Queue, the data are organized in the order of insertion**. **Both pop() and poll() retrieves the first element**. In case of ArrayDeque, the internal organization of data is dependent upon the method we are using like push(), offer().

**What is difference between LinkedList and PriorityQueue ?**

* The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
* LinkedList preserves the insertion order, PriorityQueue does not

**HashSet:** Constructs a new, empty set; the backing HashMap instance has default **initial capacity (16) and load factor (0.75**).

**Explain the differences among Enumeration, Iterator and ListIterator Enumeration**

It is a legacy interface which generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series. For example, to print all elements of a Vector<E> v:

**for (Enumeration<E> e = v.elements(); e.hasMoreElements();)**

**System.out.println(e.nextElement());**

It has only two methods, **hasMoreElements()** and **nextElement()**

**ListIterator**

It allows you to traverse in both the direction. The functionality of Enumeration interface is duplicated by the Iterator interface. **Iterator has a remove()** method while Enumeration doesn't. Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects, where as using Iterator we can manipulate the objects also like adding and removing the objects. So Enumeration is used when ever we want to make Collection objects as Read-only.

**What is the difference between Collection and Collections ?**

**Collection is an interface . Collections is a class which provides utility methods**.

Some of the significant methods are given below.

**max(Collection<? extends T> coll)** : Returns the maximum element of the given collection,

according to the natural ordering of its elements.

**max(Collection<? extends T> coll, Comparator<? super T> comp)** : Returns the maximum element of the given collection, according to the order induced by the specified comparator.

Similarly **min()** method has been given. These methods are useful when you have employee objects and you want to calculate the min and max age.

**shuffle(List<?> list)** : Randomly permutes the specified list using a default source of randomness.

**sort(List<T> list)** : Sorts the specified list into ascending order, according to the natural ordering of its elements. sort(List<T> list, Comparator<? super T> c) : Sorts the specified list according to the order induced by the specified comparator.

**What is the difference between Comparable and Comparator and when to use in situations ?**

**Comparable** is used for natural order sorting or called the default sorting. Comparator is used for custom order sorting.

Comparable interface has a method called "**compareTo(Object)**" (ie comprable to ), it has the following structure.

@Override  
 **public int** compareTo(Object o) { 🡸 **Comparable**  
 Employee emp = (Employee) o;  
 **if**(emp.**sal** == **this**.**sal**) **return** 0;  
*// else if (emp.sal < this.sal) return 1; // Ascending order* **else if** (emp.**sal** > **this**.**sal**) **return** 1; *// Descending order* **else return** -1;  
 }

**Comparator** interface has a method called "**compare(Object,Object)**" and it has the following structure.

**Ascending order**

**Comparator<Employee> ageComparator = (a,b) -> a.getAge() > b.getAge() ? 1 : -1;**

**Descending Order**

**Comparator<Employee> ageComparator = (a,b) -> a.getAge() < b.getAge() ? 1 : -1;**

Both the interface will be used in the specific situations like , we have a list of employee objects and the finance department of an organization will sort the list of employee objects based upon the salary for the income tax calculation where as other departments like HR and Personal departments will sort according to the age of the employees or based upon the employee seniority. In this type of cases, the default sorting will be based on salary for which we have to implement comparable interface. For any other custom sorting we have to use Compartor interface. Let us see a complete example.

@Data @AllArgsConstructor  
**public class** Employee **implements** Comparable {  
 **private** String **firstName**;  
 **private int sal**;  
 **private int age**;  
  
 @Override  
 **public int** compareTo(Object o) {  
 Employee emp = (Employee) o;  
 **if**(emp.**sal** == **this**.**sal**) **return** 0;  
*// else if (emp.sal < this.sal) return 1; // Ascending order* **else if** (emp.**sal** > **this**.**sal**) **return** 1; *// Descending order* **else return** -1;  
 }  
}

**public class** Test {  
 **public static void** main(String[] args) {  
 List<Employee> empList = **new** ArrayList<>();  
 empList.add(**new** Employee(**"John"**, 7000, 23));  
 empList.add(**new** Employee(**"Hati"**, 2000, 27));  
 empList.add(**new** Employee(**"Ghoda"**, 5000, 29));  
  
 Collections.*sort*(empList);  
  
 System.***out***.println(empList);  
 }  
}

**OUTPUT**

Employee(firstName=John, sal=7000, age=23),

Employee(firstName=Ghoda, sal=5000, age=29),

Employee(firstName=Hati, sal=2000, age=27)

**How to use Comparator in Java 8**

*//Ascending***Comparator<Employee> ageComparator = (a,b) -> a.getAge() > b.getAge() ? 1 : -1;**  
*//Descending***Comparator<Employee> ageComparator = (a,b) -> a.getAge() < b.getAge() ? 1 : -1;**  
Collections.*sort*(empList, ageComparator);

**Significance of toString() method**

The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

**getClass().getName() + '@' + Integer.toHexString(hashCode())**

**Difference between System.identityHashCode() and Object hashcode()**

Object hashcode gives a specific code for an object regardless of whether it is overridden or not.

**System.identityHashCode() gives the one type of pointer to the address location of an object.** If you do not override the hascode for an object both Object hashcode and System.identityHashCode() will give the same value. Let us consider an example.

public class Person {  
 private String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
  
 get()/set() methods

}

public static void main(String[] args) {  
 Person p = new Person("Deb");  
 System.*out*.println("Hashcode: " + p.hashCode());  
 System.*out*.println("System.identityHashCode: " + System.*identityHashCode*(p));  
}

**Output**

Hashcode: 796684896

System.identityHashCode: 796684896

It means that in the above case the object Person has same object hashcode and identityhascode.

If you override the hashcode of the Person class, then identity hashcode and object hascode will be different.

public class Person {  
 private String name;  
  
 public Person(String name) {  
 this.name = name;  
 }  
  
get()/set() methods  
@Override  
 public int hashCode() {  
 return Objects.*hash*(name);  
 }  
}

public static void main(String[] args) {  
 Person p = new Person("Deb");  
 *// After overiding HashCode* System.*out*.println("Hashcode: " + p.hashCode());  
 System.*out*.println("System.identityHashCode: " + System.*identityHashCode*(p));  
}

**OUTPUT**

Hashcode: 68608

System.identityHashCode: 672320506

**It means two objects are equals accoding to equals method, they have the same object hascode but may not be the same identity hashcode**. **If two objects are equal according to reference equality ie ==, they will have the same identity hashcode**. It means identity hashcode is usefull which making object comparision using == or reference equality. **IdentityHashMap internally uses Syste.identityHashCode()** or identity hashcode.

**Why 31 in hashcode ?**

Essentially the prime value is used to reduce collisions. The Java code uses 31, therefore you should use a prime other than 31 for your own calculations, such as 37.

**How JVM Handles abstract classes**

Every object has a pointer to its **vtable** in its object header. The **vtable contains pointers to all virtual and abstract methods defined in the hierarchy of the type of the object**. They are ordered and have well-known indices which makes it performant to call such a method.

**Fail Fast Vs Fail Safe Iterators in Java**

**Fail-Fast**

"Fail fast" means: it may fail ... and the failure condition is checked aggressively so that the failure condition is detected before damage can be done. **The fail-fast iterators are typically implemented using a counter (or something) on the list object**. When the list is updated, the counter is incremented. When an Iterator is created, the current value of the counter is embedded in the Iterator object. When an Iterator operation is performed, the method compares the two counter values and throws a CME if they are different.

**Fail Safe**

The only difference is fail-safe iterator doesn't throw any Exception, contrary to fail-fast Iterator, if Collection is modified structurally while one thread is iterating over it. This is because they work on clone of Collection instead of original collection and that’s why they are called as **fail-safe iterator**. **Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMap keySet is also fail-safe iterator and never throw ConcurrentModificationException in Java**.

But in this approach we can find the following two issues:

a. The overhead of copying the data structure (time and memory)

b. The fail-safe iterator doesnt guarantees the data being read is the data currently in the data structure with the created collection.

**Array vs linked list in Java**

* LinkedList does not support random access while Array supports
* Array needs contiguous memory allocation, which may result in java.lang.OutOfMemoryError.
* Linked list is distributed data structure, it's element are scattered over heap and doesn't need a contiguous memory allocation.
* Array is a fixed length data structure.
* Inserting element at beginning of linked list, and deleting element from end of linked list is O(1) operation
* Array is ideal for implementing fast caches e.g. HashMap or Hashtable, which requires constant time retrieval.
* Array can be one or multi-dimensional, while linked list can be singly, doubly or circular linked list

**LinkedList vs ArrayList in Java and When to use which one**

* **ArrayList provides constant time for search operation, so it is better to use ArrayList if searching is more frequent operation than add and remove operation**.
* The LinkedList provides constant time for add and remove operations. So it is better to use LinkedList for manipulation.
* **For frequent insertion and deletion, LinkedList is good** and there is no resizing of array.
* LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

**LinkedList**

* **get(int index) is O(n)** (with n/4 steps on average), but O(1) when index = 0
* **add(int index, E element) is O(n)** (with n/4 steps on average), but O(1) when index = 0
* **remove(int index) is O(n)** (with n/4 steps on average), but O(1) when index = 0
* **Iterator.remove() is *O(1)***. **One of the main benefits of** LinkedList<E>
* **ListIterator.add(E element) is *O(1)***. **One of the main benefits of** LinkedList<E>
* **LinkedList<E> allows for constant-time insertions or removals using iterators**

**ArrayList**

* **get(int index) is *O(1)***. **Main benefit of** ArrayList<E>
* **add(E element) is *O(1)***amortized, but *O(n)* worst-case since the array must be resized and copied
* add(int index, E element) is *O(n)* (with *n/2* steps on average)
* remove(int index) is *O(n)* (with *n/2* steps on average)
* Iterator.remove() is *O(n)* (with *n/2* steps on average)
* ListIterator.add(E element) is *O(n)* (with *n/2* steps on average)

**Amortized meaning: Gradual increase.**

**What is Red-Black Tree Algorithm: 🡺 TreeMap uses Red-Black Tree Algorithm 🡸**

In computer science, a red–black tree is a kind of self-balancing binary search tree. **Each node stores an extra bit representing "color" ("red" or "black"), used to ensure that the tree remains balanced during insertions and deletions**.

**Rules That Every Red-Black Tree Follows:**

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
5. All leaf nodes are black nodes.

Search 🡺 O(log n)

Insert 🡺 O(log n) 🡺 **n** is the total number of elements in the red-black tree

Delete 🡺 O(log n)

**Concurrent Java DataStructure**

**Class ConcurrentLinkedQueue<E>**: An unbounded thread-safe [queue](file:///E:\dev\java7\docs\api\java\util\Queue.html) based on linked nodes. This queue orders elements FIFO (first-in-first-out).

**Class ConcurrentLinkedDeque<E>:** An unbounded concurrent [deque](file:///E:\dev\java7\docs\api\java\util\Deque.html) based on linked nodes. Concurrent insertion, removal, and access operations execute safely across multiple threads.

**Class ConcurrentSkipListMap<K,V>**: A scalable concurrent [ConcurrentNavigableMap](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentNavigableMap.html) implementation. The map is sorted according to the [natural ordering](file:///E:\dev\java7\docs\api\java\lang\Comparable.html) of its keys, or by a [Comparator](file:///E:\dev\java7\docs\api\java\util\Comparator.html) provided at map creation time, depending on which constructor is used. **Threadsafe variant of TreeMap**

**Class ConcurrentSkipListSet<E>**: A scalable concurrent [NavigableSet](file:///E:\dev\java7\docs\api\java\util\NavigableSet.html) implementation based on a[**ConcurrentSkipListMap**](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListMap.html)**.** The elements of the set are kept sorted according to their [natural ordering](file:///E:\dev\java7\docs\api\java\lang\Comparable.html), or by a [Comparator](file:///E:\dev\java7\docs\api\java\util\Comparator.html) provided at set creation time, depending on which constructor is used**. ThreadSafe variant of TreeSet.**

**Class CopyOnWriteArrayList<E>**: A thread-safe variant of [ArrayList](file:///E:\dev\java7\docs\api\java\util\ArrayList.html) in which all mutative operations (add, set, and so on) are implemented by making a fresh copy of the underlying array**.**

**Class CopyOnWriteArraySet<E>**: A [Set](file:///E:\dev\java7\docs\api\java\util\Set.html) that uses an internal [CopyOnWriteArrayList](file:///E:\dev\java7\docs\api\java\util\concurrent\CopyOnWriteArrayList.html) for all of its operations. Threadsafe variant of HashSet.

**Class LinkedBlockingQueue<E>**: An optionally-bounded [blocking queue](file:///E:\dev\java7\docs\api\java\util\concurrent\BlockingQueue.html) based on linked nodes. This queue orders elements FIFO (first-in-first-out).

**Interface NavigableSet:**It may be accessed and traversed in either ascending or descending order. The descendingSet method returns a view of the set with the senses of all relational and directional methods inverted. The performance of ascending operations and views is likely to be faster than that of descending ones.

**How RandomAccess interface is used as MarkerInterface?**

**Refer to the class AbstractList.java**

**public** List<E> subList(**int** fromIndex, **int** toIndex) {  
 **return** (**this instanceof** RandomAccess ?  
 **new** RandomAccessSubList<>(**this**, fromIndex, toIndex) :  
 **new** SubList<>(**this**, fromIndex, toIndex));

Consider a real-life scenario: In a hospital, patients are attended to based on the severity of their conditions, not just their arrival time. This is a classic real-world demonstration of a Priority Queue.

If you need high performance for enqueue and dequeue operations and do not require null elements, ArrayDeque is typically the better choice.