Complete Guide to Java Priority Queue

• Azhwani • Nov 22, 2021 • java • 7 mins read

Table of Contents

In this article, we are going to cover in-depth the **priority queue in Java**.

First, we will start with a little bit of theory about what a priority queue is. Then, we will go down the rabbit hole to see how to use *PriorityQueue* in practical examples.

What is Priority Queue in Java

In short, a priority queue is a special unbounded type of queue where all elements are ordered by priority.

The priority of each element is calculated **according to the natural ordering, or using a custom comparator**.

In Java, a priority queue is represented by the *PriorityQueue* class. This class belongs to Java's collections API and implements the *Queue* interface.

The following are some of the main points keys of the *PriorityQueue* class:

- The head of a priority queue is the least element according to the specified ordering
- By default, *PriorityQueue* sorts elements based on their natural order
- *PriorityQueue* does not allow *null*
- *PriorityQueue* can't store non-comparable objects, otherwise a *ClassCastException* will be raised

Please bear in mind that *PriorityQueue* is not synchronized in Java.

So, in a multi-thread environment, we need to make sure that multiple threads do not access the priority queue while one of the other threads modifies it.

Priority Queue Example

Now that we know what the priority queue is in Java, let's see how we can create and manipulate a basic *PriorityQueue*.

For instance, let's consider a priority queue of integers:

```
    public static void main(String[] args) {
        PriorityQueue<Integer> queueIntegers = new PriorityQueue<Integer>();
        queueIntegers.add(568);
        queueIntegers.add(123);
        queueIntegers.add(895);
        queueIntegers.add(10);
        queueIntegers.add(785);
        while (!queueIntegers.isEmpty()) {
            System.out.println(queueIntegers.poll());
```

```
        }
    }
    # Output:
    10
    123
    568
    785
    895
```

As we can see, the smallest integer appears at the head of the queue and is removed first.

poll() retrieves the latest element and also removes it from the queue. However, if we want to get the element without removing it, we can use *peek()*.

Now, let's create a *PriorityQueue* of *String* objects and see how they will be removed:

```
    public static void main(String[] args) {
        PriorityQueue queueOfCities = new PriorityQueue<String>();
        queueOfCities.add("Tamassint");
        queueOfCities.add("Madrid");
        queueOfCities.add("Paris");
        queueOfCities.add("Rio de Janeiro");
        queueOfCities.add("New York");
        while (!queueOfCities.isEmpty()) {
            System.out.println(queueOfCities.poll());
        }
    }
    # Output:
    Madrid
    New York
    Paris
    Rio de Janeiro
    Tamassint
```

As shown above, *PriorityQueue* fetches and removes *String* elements alphabetically in ascending order.

Please bear in mind that if we don't provide the priority queue with an order, it will prioritize its items according to the default natural ordering.

PriorityQueue with Reverse Order

Now, let's see how to sort priority queue elements in reverse order.

Fortunately, the *Collections* class provides a convenient way to do that using the *reverseOrder()* method.

We can pass *Collections.reverseOrder()* to *PriorityQueue* constructor:

```
public static void main(String[] args) {
    PriorityQueue queueOfCountries = new PriorityQueue(Collections.reverseOrder());
    queueOfCountries.add("Malaysia");
    queueOfCountries.add("Jamaica");
    queueOfCountries.add("Belgium");
    queueOfCountries.add("United Kingdom");
    queueOfCountries.add("Australia");
    while (!queueOfCountries.isEmpty()) {
        System.out.println(queueOfCountries.poll());
    }
}
# Output:
United Kingdom
Malaysia
Jamaica
Belgium
Australia
```

As shown above, the priority queue has done its job well and prioritized its elements in reverse order.

PriorityQueue of User Defined Objects

Up to this point, we have seen how a Java priority queue orders strings and integers.

So, in this chapter, we will showcase how to create a *PriorityQueue* of custom Java objects.

PriorityQueue needs to compare its objects to sort them accordingly.

So, we need to provide a custom *Comparator* that defines the order logic or force our user-defined class to implement the *Comparable* interface.

Otherwise, the priority queue will simply throw a *ClassCastException* exception.

Let's suppose we have a Java class called *Product*:

```
public class Product implements Comparable<Product> {
    private String name;
    private double price;
    public Product(String name, double price) {
        super();
        this.name = name;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
    public int compareTo(Product o) {
        // Write your order logic here
    }
}
```

As shown above, our *Product* class implements *Comparable* interface and overrides the *compareTo* method.

Simply put, the code specified in *compareTo* method will decide the ordering that will be used to prioritize elements.

Let's say we want to order our *Product* objects by thier names:

```
public int compareTo(Product o) {
    if(o == null) {
        return 1;
```

```
        }
        return getName().compareTo(o.getName());
    }
```

getName().compareTo(o.getName()) instructs that the products should be prioritized lexicographically based on their names.

Now, let's create a priority queue and add some *Product* objects to it:

```
    public static void main(String[] args) {
        PriorityQueue queueOfProducts = new PriorityQueue<Product>();
        queueOfProducts.add(new Product("iPhone 13 Pro Max", 1099.99));
        queueOfProducts.add(new Product("galaxy S21 5G", 799.99));
        queueOfProducts.add(new Product("xP3plus", 210));
        queueOfProducts.add(new Product("9 Pro 5G", 899.99));
        queueOfProducts.add(new Product("moto g stylus 5G", 252));
        while (!queueOfProducts.isEmpty()) {
            System.out.println(queueOfProducts.poll().getName());
        }
    }
    # Output:
    9 Pro 5G
    galaxy S21 5G
    iPhone 13 Pro Max
    moto g stylus 5G
    xP3plus
```

As we can see, the products are sorted lexicographically in ascending order.

The limitation of the *Comparable* interface is that it provides only one version of order. So, we need to change the *compareTo* implementation each time we want to apply a new order logic.

To address this limitation we can use different implementations of the *Comparator* interface.

Priority Queue with Comparator

Unlike *Comparable*, we can implement *Comparator* interface outside our classes. That way, we can create multiple comparators, each one defines one specific order.

Once we create our comparators, we can pass the version we want to our priority queue during construction.

Using Custom Comparator

To create a custom comparator, we need to implement the *Comparator* interface and override the *compare* method.

For example, let's create a comparator to sort our products in descending order based on the *price* property:

```
public class ProductPriceComparator implements Comparator<Product> {
    // Descending Order
    public int compare(Product o1, Product o2) {
        return o1.getPrice() < o2.getPrice() ? 1 : -1;
    }
}
```

For instance, let's add some *Product* instances to our priority queue and retrieves them:

```
public static void main(String[] args) {
    PriorityQueue queueOfProducts = new PriorityQueue<Product>();
    queueOfProducts.add(new Product("Hp ENVY x360", 900));
    queueOfProducts.add(new Product("Acer Predator 17", 799.99));
    queueOfProducts.add(new Product("Lenovo ThinkPad X250", 999.99));
    queueOfProducts.add(new Product("Dell Alienware 18inch", 950));
    while (!queueOfProducts.isEmpty()) {
        Product product = queueOfProducts.poll();
        System.out.println(product.getName() + " : " + product.getPrice() + "$");
    }
}
# Output:
Acer Predator 17 : 799.99$
Dell Alienware 18inch : 950.0$
Hp ENVY x360 : 900.0$
Lenovo ThinkPad X250 : 999.99$
```

Now, if we want to use another custom order, all we need to do is create another comparator and pass it to our priority queue.

So, let's do this but with the help of the built-in Java 8 comparators.

Using Java 8 Comparator

Java 8 takes comparators creation to the next level. With the new enhancements, we don't have to create a separate class to implement a custom comparator.

We can simply use one of the multiple variants of the *Comparator.comparing* static function.

Let's tell our java priority queue to order the products in ascending order according to thier *price*:

```
public static void main(String[] args) {
    Comparatorspan><Product> productPriceComparator = Comparator
        .comparingDouble(Product::getPrice);

    PriorityQueue queueOfProducts = new PriorityQueue<Product>(productPriceComparator);
    queueOfProducts.add(new Product("iPad Mini 256GB", 360));
    queueOfProducts.add(new Product("iPad Pro 11-inch, 128GB", 800));
    queueOfProducts.add(new Product("iPad Pro 12.9-inch, 128GB", 1099));
    queueOfProducts.add(new Product("iPad Air 256GB", 749));
    while (!queueOfProducts.isEmpty()) {
        Product product = queueOfProducts.poll();
        System.out.println(product.getName() + " : " + product.getPrice() + "$");
    }
}
# Output:
iPad Mini 256GB : 360.0$
iPad Air 256GB : 749.0$
iPad Pro 11-inch, 128GB : 800.0$
iPad Pro 12.9-inch, 128GB : 1099.0$
```

Let's now create another comparator to tell our *PriorityQueue* to sort the products by *name* in descending order:

```
Comparator<Product> productNameReversedComparator = Comparator
    .comparing(Product::getName)
```

```
            .reversed();
```

Cool, right?

As we can see, Java 8 provides a more fluent and concise way to create custom comparators.

Using Lambda Expression

Lambda expression offers another simple and concise syntax for creating custom comparators.

With the help of Lambda expression, we can pass a comparator directly to our priority queue to indicate how it will prioritize its Java objects.

Let's see how to tell *PriorityQueue* to prioritize *Product* instances based on thier name length:

```
Comparator<Product> productNameLengthComparator =
    (Product o1, Product o2) -> o1.getName().length() - o2.getName().length();
PriorityQueue myProductsQueue= new PriorityQueue(productNameLengthComparator);
```

Conclusion

In this short tutorial, we have explained what a Java priority queue is.

Along the way, we have seen how to create a priority queue with custom Java objects.

Then, we showcased how to use the *Comparable* and *Comparator* interfaces to tell *PriorityQueue* how to prioritize its elements.

Liked the Article? Share it on Social media!

Facebook
Linkedin
Reddit

If you enjoy reading my articles, buy me a coffee ☕. I would be very grateful if you could consider my request ✌️