Core Java- Xtra -2017

# PriorityQueue

<http://stackoverflow.com/questions/683041/how-do-i-use-a-priorityqueue>

Use the constructor overload which takes a Comparator<? super E> comparator and pass in a comparator which compares in the appropriate way for your sort order. If you give an example of how you want to sort, we can provide some sample code to implement the comparator if you're not sure. (It's pretty straightforward though.)

As has been said elsewhere: offer and add are just different interface method implementations. In the JDK source I've got, add calls offer. Although add and offer have *potentially* different behaviour in general due to the ability for offer to indicate that the value can't be added due to size limitations, this difference is irrelevant in PriorityQueue which is unbounded.

Here's an example of a priority queue sorting by string length:

import java.util.Comparator;

import java.util.PriorityQueue;

public class Test {

public static void main(String[] args) {

Comparator<String> comparator = new StringLengthComparator();

PriorityQueue<String> queue =

new PriorityQueue<String>(10, comparator);

queue.add("short");

queue.add("very long indeed");

queue.add("medium");

while (queue.size() != 0) {

System.out.println(queue.remove());

}

}

}

// StringLengthComparator.java

import java.util.Comparator;

public class StringLengthComparator implements Comparator<String> {

@Override

public int compare(String x, String y) {

// Assume neither string is null. Real code should

// probably be more robust

// You could also just return x.length() - y.length(),

// which would be more efficient.

if (x.length() < y.length()) {

return -1;

}

if (x.length() > y.length()) {

return 1;

}

return 0;

}

}

Here is the output:

short

medium

very long indeed

When you add or remove elements from PriorityQueue, other elements are compared to each other to put highest/lowest priority element at the head of the queue.  priority queue data structure is internally implemented using binary heap data structure, which allows constant time access to the maximum and minimum element in a heap by implementing max heap and min heap data structure.

# Why Clone method is protected

<http://stackoverflow.com/questions/1138769/why-is-the-clone-method-protected-in-java-lang-object>

The fact that clone is protected is extremely dubious - as is the fact that the clone method is not declared in the Cloneable interface.

It makes the method pretty useless for taking copies of data because **you cannot say**:

if(a instanceof Cloneable) {

copy = ((Cloneable) a).clone();

}

I think that the design of Cloneable is now **largely regarded as a mistake** (citation below). I would normally want to be able to make implementations of an interface Cloneable but *not necessarily make the interface Cloneable* (similar to the use of Serializable). This cannot be done without reflection:

ISomething i = ...

if (i instanceof Cloneable) {

//DAMN! I Need to know about ISomethingImpl! Unless...

copy = (ISomething) i.getClass().getMethod("clone").invoke(i);

}

Citation From **Josh Bloch's Effective Java**:  
*"The Cloneable interface was intended as a mixin interface for objects to advertise that they permit cloning. Unfortunately it fails to serve this purpose ... This is a highly a typical use of interfaces and not one to be emulated ... In order for implementing the interface to have any effect on a class, it and all of its superclasses must obey a****fairly complex, unenforceable and largely undocumented protocol****"*

The Clonable interface is just a marker saying the class can support clone. The method is protected because you shouldn't call it on object, you can (and should) override it as public.

From Sun:

In class Object, the clone() method is declared protected. If all you do is implement Cloneable, only subclasses and members of the same package will be able to invoke clone() on the object. To enable any class in any package to access the clone() method, you'll have to override it and declare it public, as is done below. (When you override a method, you can make it less private, but not more private. Here, the protected clone() method in Object is being overridden as a public method.)

clone is protected because it is something that ought to be overridden so that it is specific to the current class. While it would be possible to create a public clone method that would clone any object at all this would not be as good as a method written specifically for the class that needs it.

<http://stackoverflow.com/questions/31314468/why-object-clone-method-is-protected-not-public>

**You don't always want your objects to be cloneable. Sometimes you have classes that represent something that should not be cloneable**.

Unfortunately, the designers of the Java language used an odd design when they added this feature to class Object. There's a protected clone() method which throws a CloneNotSupportedException, unless the class implements interface Cloneable. **Interface Cloneable is a marker interface (an interface without any methods - its only purpose is to indicate that the class that implements it is allowed to be cloned).** If you want, you can override the clone() method in your class and make it public.

<http://stackoverflow.com/questions/18240037/why-object-clone-method-available-only-to-classes-that-implement-cloneable-int>

Object's clone() method is quite special, as it always returns an instance of the current class that has all fields copied (even final). I don't think its possible to reproduce this with plain Java code, not even with reflection.

Because of this, it must be made available to all classes, but since it should not be callable from the outside by default because you don't want everything to be cloneable, it must be protected.

As an additional check, clone checks that the class implements Cloneable, only to ensure you don't clone non-cloneables by accident.

<https://www.quora.com/Why-does-java-Object-class-have-protected-Object-clone-throws-CloneNotSupportedException-rather-than-public-Object-clone-throws-CloneNotSupportedException>

Since subclasses inherit public instance methods and access to inherited methods cannot be decreased, if Object declared clone() as public, then every class would have to have a public clone() method. This would pollute the api of every class regardless of whether it could actually be cloned or not.