11.4 Comparators

We've just learned about the comparable interface, which imbeds into each Dog the ability to compare itself to another Dog. Now, we will introduce a new interface that looks very similar called `Comparator`.

Let's start off by defining some terminology.

- Natural order - used to refer to the ordering implied in the `compareTo` method of a particular class.

As an example, the natural ordering of Dogs, as we stated previously, is defined according to the value of size. What if we'd like to sort Dogs in a different way than their natural ordering, such as by alphabetical order of their name?

Java's way of doing this is by using `Comparator`'s. Since a comparator is an object, the way we'll use `Comparator` is by writing a nested class inside Dog that implements the `Comparator` interface.

But first, what's inside this interface?

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

This shows that the `Comparator` interface requires that any implementing class implements the `compare` method. The rule for `compare` is just like `compareTo`:

- Return negative number if o1 < o2.
- Return 0 if o1 equals o2.
- Return positive number if o1 > o2.

Let's give Dog a NameComparator. To do this, we can simply defer to `String`'s already defined `compareTo` method.

```
import java.util.Comparator;

public class Dog implements Comparable<Dog> {
    ...
    public int compareTo(Dog uddaDog) {
        return this.size - uddaDog.size;
    }

    private static class NameComparator implements Comparator<Dog> {
        public int compare(Dog a, Dog b) {
            return a.name.compareTo(b.name);
        }
    }

    public static Comparator<Dog> getNameComparator() {
        return new NameComparator();
    }
}
```

Note that we've declared NameComparator to be a static class. A minor difference, but we do so because we do not need to instantiate a Dog to get a NameComparator. Let's see how this Comparator works in action.

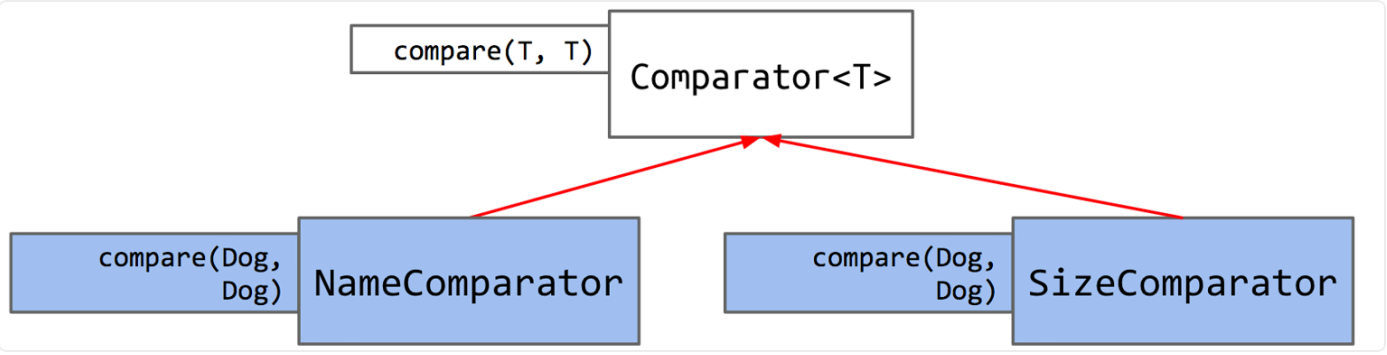
[Inheritance3, Video 6] Comparator

As you've seen, we can retrieve our NameComparator like so:

```
Comparator<Dog> nc = Dog.getNameComparator();
```

All in all, we have a Dog class that has a private NameComparator class and a method that returns a NameComparator we can use to compare dogs alphabetically by name.

Let's see how everything works in the inheritance hierarchy - we have a Comparator interface that's built-in to Java, which we can implement to define our own Comparators (`NameComparator` , `SizeComparator` , etc.) within Dog.



To summarize, interfaces in Java provide us with the ability to make **callbacks**. Sometimes, a function needs the help of another function that might not have been written yet (e.g. `max` needs `compareTo`). A callback function is the helping function (in the scenario, `compareTo`). In some languages, this is accomplished using explicit function passing; in Java, we wrap the needed function in an interface.

A Comparable says, "I want to compare myself to another object". It is imbedded within the object itself, and it defines the **natural ordering** of a type. A Comparator, on the other hand, is more like a third party machine that compares two objects to each other. Since there's only room for one `compareTo` method, if we want multiple ways to compare, we must turn to Comparator.

Previous
11.3 Comparables

Next
11.5 Chapter Summary

Last updated 1 year ago