SHORTCUT

Filtering and Predicates

Hi. My name is Johannes Dvorak Lagos. I work as senior mobile developer at Shortcut here in Oslo. Today I will talk to you about how to use predicates for filtering collections on Android. I will also cover a little bit sorting of collections.

First I want to present too you a data model. In this occasion it is a person. Normally a person have first and last name, an age, an address, perhaps a place to work at, maybe a family, etc. Could be many possibilities. This could be any data model you are working at.

There will be very few times that you will work on a very simple dataset, like just one object. Normally you will have much more complex models.
For this example I will use 3 objects, of course one for person, one for address, and work will also be an object. However I will not show you live coding this time since we only have 10 minutes, therefor I will resume the coding on a few slides.

SHORTCUT

```
List<Address> sortedByCity = new ArrayList<>();
List<Person> peopleSortedByCity = new ArrayList<>();
for (Person p : listOfPerson) {
    sortedByCity.add(p.address);
}

Collections.sort(sortedByCity);
for (Address a : sortedByCity) {
    for (Person p : listOfPerson) {
        if (p.address.equals(a)) {
            peopleSortedByCity.add(p);
        }
    }
}
```

So, different situations will give different need of data or filtering your the existing data. Perhaps you will have to do some special sorting, for giving you an example, could be sorting by city descending, then by first name and by last name in case of equals first names. If you develop only for the SDK 24 you will have the support given by Java 8 predicates, if not you have to use 3rd parties libraries or do it old school.

In this example I used the Comparable interface on the address object, so I could on a easy view sort by city from the inside of the object itself. Then iterated through the list of persons and find the existing city where the persons lived, then sort the list of cities, after that I iterated through the sorted city list and picked out persons that belong to that city. Not an elegant solution, right?

```
public int compareTo(Person o) {
    return this.address.city.compareTo(o.address.city);
}
```

Instead, you could use the Comparable interface inside the Person object for sorting by cities. However, if you want to sort by many attributes you would have a problem because sorting with the Comparable interface doesn’t allow it.

SHORTCUT

```
public enum ObjectComparator implements Comparator<Person> {
    ORDER_BY_FIRSTNAME {
        public int compare(Person o1, Person o2){
            return o1.firstName.compareTo(o2.firstName);
        }
    },
    ORDER_BY_CITY_DESCENDING {
        public int compare(Person o1, Person o2){
            return -1 * (o1.address.city.compareTo(o2.address.city));
        }
    },
    ORDER_BY_LASTNAME {
        public int compare(Person o1, Person o2){
            return o1.lastName.compareTo(o2.lastName);
        }
    };

    public static Comparator<Person> compareWithSetOfComparator(final ObjectComparator...comparators){
        return new Comparator<Person>() {
            @Override
            public int compare(Person o1, Person o2) {
                for(ObjectComparator option: comparators){
                    int result = option.compare(o1, o2);
                    if(result != 0){
                        return result;
                    }
                }
                return 0;
            }
        };
    }
}
```

One way to solve it could be using enum with different sorting algorithms and a static method that collect several comparators. Then you can initiate Comparator object from the ObjectComparator and the set of needed sorting. For our case it will mean that we call the order_by_city descending, the order by first name, and at the end last name. You could also see that the latest two is ordering on a natural way. With this solution you will be able to fulfill the expectation of given sorting. But it still doesn’t look that elegant.

```
Predicate<Person> prediction = new Predicate<Person>() {
    @Override
    public boolean apply(Person input) {
        return input.address.country.equals("USA");
    }
};
Iterable<Person> personIterable = Iterables.filter(listOfPerson, prediction);
```

As you imagine, filtering is not that «simple» with java. You have to use for loops, initiate a sublist, etc. Or you could use SQLite own selected sentence and let the database return filtered data sets.

When I was developing iOS apps I came over NSPredicate. Actually I was kind of jealous. Why doesn't that exist for Java?. I will not go into the technical details of predicate, but as the name say, you use a prediction for filtering. As I told you earlier, predicate on Java exist only in the Java 8 SDK, for android that means that you have to only use the latest sdk. Or you could use Googles library called Guava. Guava was Google own initiative to include stuffs that was not part of java language.

In this example the predicate will filter all the person from USA. It looks quite easy, right. The problem with Guava is how huge the library it is if you don't use proguard. Last time I checked it was around 15k. Anyway, Google is splitting up this mastodont into smaller libraries. That will means that you don't need to import the whole library for using predicates.

You also have Kotlin. Kotlin, is Jetbrains the creator of Intellij own language, it have the strength that include features in the language that Java 6 & 7 lack. It will add some methods to your project, I think is about 9k. But it have a lot of snacks that make you avoid another libraries, and you will get rid of tons of boiler plate code. Like in this case the predicate is just one line, removing the verbosity of Java & Guava. The filtering is also happening inside a lambda. The filtering method is part of the Collections interface. So you will have access to it even for filtering strings.

SHORTCUT

```
return listOfPerson.sortedBy { it.lastName }.sortedBy { it.firstName }.sortedByDescending
{ it.address.city }
```

To finish, do you remember the enum for sorting by city, first name and last name? Here is the Kotlin answer. Kotlin allow chained methods. Also when it comes to sortedBy methods. As you see it is much more readable and understandable than the enum with the collection of comparators I showed you for some slides ago.

SHORTCUT

Johannes Dvorak Lagos

Senior Mobile Developer
johannes.lagos@shortcut.no

Thank you for listening.