# Object-Oriented Sets: Instructions

When you're ready to submit your solution, go to the assignments list.

Download the objsets.zip handout archive file.

In this assignment you will work with an object-oriented representations based on binary trees.

# Object-Oriented Sets

For this part, you will earn credit by completing the `TweetSet.scala` file. This file defines an abstract class `TweetSet` with two concrete subclasses, `Empty` which represents an empty set, and `NonEmpty(elem: Tweet, left: TweetSet, right: TweetSet)`, which represents a non-empty set as a binary tree rooted at `elem`. The tweets are indexed by their text bodies: the bodies of all tweets on the `left` are lexicographically smaller than `elem` and all bodies of elements on the `right` are lexicographically greater.

Note also that these classes are *immutable*: the set-theoretic operations do not modify `this` but should return a new set.

Before tackling this assignment, we suggest you first study the already implemented methods `contains` and `incl` for inspiration.

## 1 Filtering

Implement filtering on tweet sets. Complete the stubs for the methods `filter` and `filter0`. `filter` takes as argument a function, the predicate, which takes a tweet and returns a boolean. `filter` then returns the subset of all the tweets in the original set for which the predicate is true. For example, the following call:

```
tweets.filter(tweet => tweet.retweets > 10)
```

applied to a set `tweets` of two tweets, say, where the first tweet was not retweeted and the second tweet was retweeted 20 times should return a set containing only the second tweet.

Hint: start by defining the helper method `filter0` which takes an accumulator set as a second argument. This accumulator contains the ongoing result of the filtering.

```
/** This method takes a predicate and returns a subset of all the elements
 *  in the original set for which the predicate is true.
 */
def filter(p: Tweet => Boolean): TweetSet
def filter0(p: Tweet => Boolean, accu: TweetSet): TweetSet
```

The definition of `filter` in terms of `filter0` should then be straightforward.

## 2 Taking Unions

Implement union on tweet sets. Complete the stub for the method `union`. The method `union` takes another set `that`, and computes a *new* set which is the union of `this` and `that`, i.e. a set that contains exactly the elements that are *either* in `this` *or* in `that`, *or in both*.

```
def union(that: TweetSet): TweetSet
```

Note that in this exercise it is your task to find out in which class(es) to define the `union` method (should it be abstract in class `TweetSet`?).

Hint: you have methods `head`, `tail` on `TweetSet` available to you that would be helpful for this part. They're analogous to the same methods on `List`. That is, `head` returns one element of the set, while `tail` returns a set with all elements of the original set, except for the `head` element.

## 3 Sorting Tweets by Their Influence

The more often a tweet is "re-tweeted" (that is, repeated by a different user with or without additions), the more influential it is.

The goal of this part of the exercise is to add a method `ascendingByRetweet` to `TweetSet` which should produce a linear sequence of tweets (as an instance of class `Trending`), ordered by their number of retweets:

```
def ascendingByRetweet: Trending
```

This method reflects a common pattern when transforming data structures. While traversing one data structure (in this case, a `TweetSet`), we're building a second data structure (here, an instance of class `Trending`). The idea is to start with an instance of `EmptyTrending` (containing no tweets), and to find the tweet with the fewest retweets in the input `TweetSet`. This tweet is removed from the `TweetSet` (that is, we obtain a new `TweetSet` that has all the tweets of the original set except for the tweet that was "removed"; this *immutable* set operation, `remove`, is already implemented for you), and added to the trending, producing a new `NonEmptyTrending`. After that, the process repeats itself, but now we are searching through a `TweetSet` with one less tweet.

Hint: use the already-implemented `findMin` and `remove` methods and the earlier-introduced "accumulator" pattern.

## 4 Tying everything together

In the last step of this assignment your task is to detect influential tweets in a set of recent tweets. We are providing you with a `TweetSet` containing several hundred tweets from popular tech news sites in the past few days, located in the `TweetReader` object (file “TweetReader.scala”). `TweetReader.allTweets` returns an instance of `TweetSet` containing a set of all available tweets.

Furthermore, you are given two lists of keywords. The first list corresponds to keywords associated with Google and Android smartphones, while the second list corresponds to keywords associated with Apple and iOS devices. Your objective is to detect which platform has generated more interest or activity in the past few days.

As a first step, use the functionality you implemented in the first parts of this assignment to create two different `TweetSet`s, `googleTweets` and `appleTweets`. The first `TweetSet`, `googleTweets`, should contain all tweets that mention (in their “text”) one of the keywords in the `google` list. The second `TweetSet`, `appleTweets`, should contain all tweets that mention one of the keyword in the `apple` list. Their signature is as follows:

```
val googleTweets: TweetSet
val appleTweets: TweetSet
```

Hint: use the `exists` method of `List` and `contains` method of class `java.lang.String`.

From the *union* of those two `TweetSet`s, produce `trending`, an instance of class `Trending` representing a sequence of tweets ordered by their number of retweets:

```
val trending: Trending
```