Chapter 9 – Maps

Contents

[1 Maps 2](#_Toc131507740)

[2 The *HashMap* Class 3](#_Toc131507741)

[2.1 Exercises 5](#_Toc131507742)

[3 Example 1: Map of Employee Objects 6](#_Toc131507743)

[4 Example 2: Map of Lists 7](#_Toc131507744)

[5 Example 3: 1 to Many 8](#_Toc131507745)

[6 The *LinkedHashMap* Class 11](#_Toc131507746)

[7 The *TreeMap* Class 11](#_Toc131507747)

[7.1 Exercises 12](#_Toc131507748)

[8 Example 4: Occurrences of Words & Modelling 13](#_Toc131507749)

[8.1 Problem Description 13](#_Toc131507750)

[8.2 Algorithm 13](#_Toc131507751)

[8.3 Method 14](#_Toc131507752)

[8.4 Single Responsibility Principle 14](#_Toc131507753)

[8.5 Refactoring for Reuse 15](#_Toc131507754)

[8.6 Exercises 18](#_Toc131507755)

[Appendix 1 The *SortedMap* & *NavigableMap* Interfaces 19](#_Toc131507756)

[Appendix 2 Class Diagram for Map Classes 20](#_Toc131507757)

# Maps

A *map* stores objects, just like a list. However, instead of associating an index with each object, a *map* associates a unique *key* with each object. is like a list. In computing, we say that a map stores *key-value pairs*. For example:

* Banner: a student number (key) is associated with each student’s records (value).
* Web Server: a URL (key) is associated with each web page (value).
* Dictionary: each word (key) is associated with a definition (value)
* Phone book: each name (key) is associated with a phone number (value).

You can add, retrieve, and remove from a map with code like this:

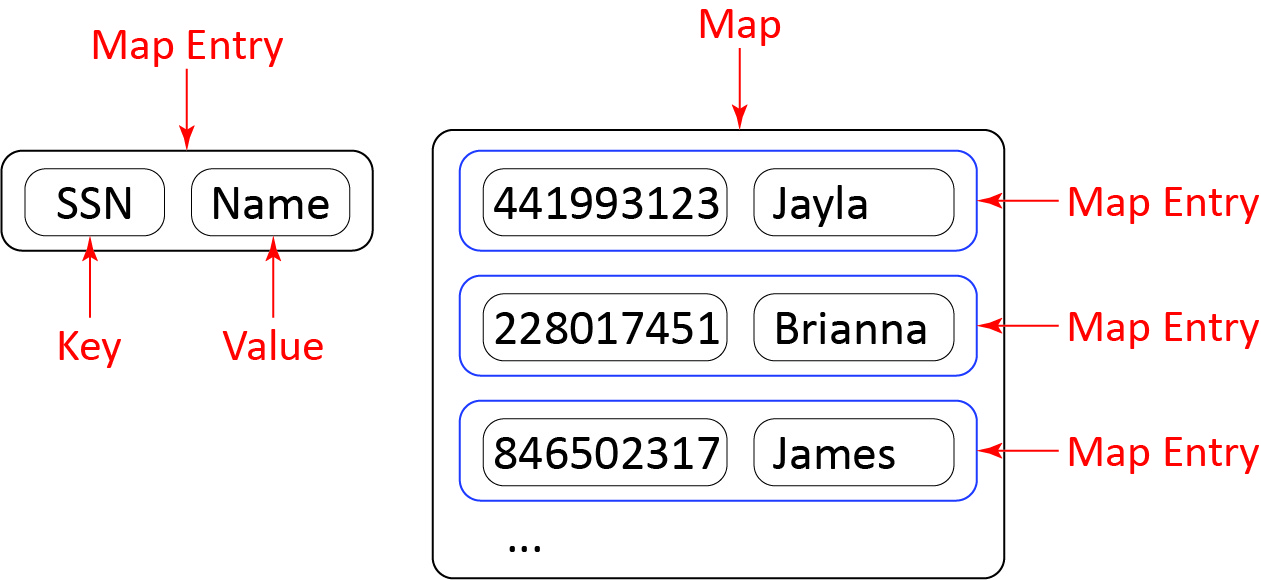
map.put(key,object)

object = map.get(key)

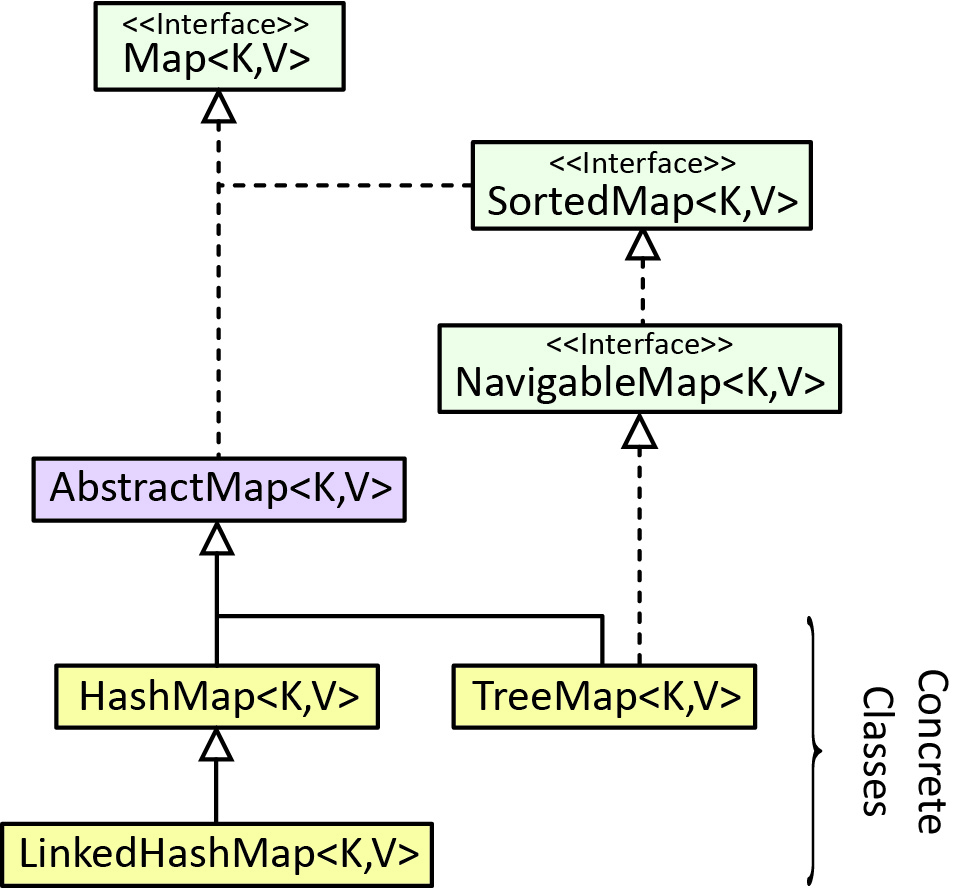
object = map.remove(key)

Maps are very useful.

Java refers to key-value pairs as *map entries.* The *Map* interface is designed to store *map entries.* The example on the left below shows a *map entry* that consists of a *key* (person’s SSN)and a *value* (person’s name). The figure on the right below shows a *map* with a number of map entries. The keys must be unique, in other words, there are no duplicate keys in a map. We would describe this map this way: *a map of names where SSN is the key*.



The *Map* interface is part of the Java Collections Framework[[1]](#footnote-1) (JCF). As shown in the diagram below, Java provides three common implementations of the *Map* interface. The main difference is the order the items are stored in.

1. *HashMap* – The map entries have no particular order.
2. *LinkedHashMap* – The map entries are ordered according to the order they were inserted.
3. *TreeMap* – The map entries are ordered according to their keys.

Note that the classes have two generic type parameters, K & V. Thus, you must specify the reference type for the key and the value. From the example above, the map of names:

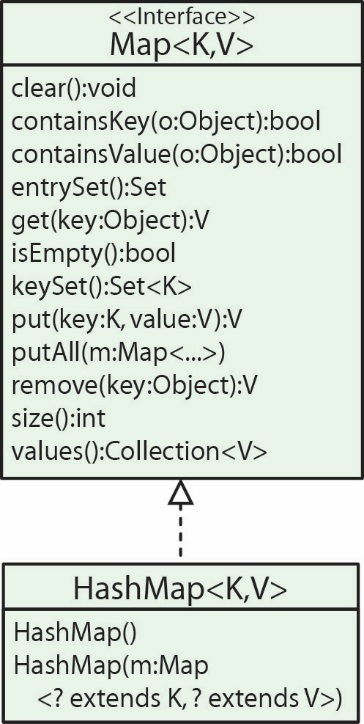
HashMap<Integer,String> names = new HashMap<>()

names.put(754891113, “Stan”)

String name = names.get(228017451)

# The *HashMap* Class

The example in this section is in the *example\_name\_score* package.

Some of the methods defined in the *Map* interface are shown in the class diagram on the right, where *HashMap* is a concrete implementation of *Map*. As stated earlier, a *HashMap* does not allow duplicate keys and the order of the elements is not predictable. A *HashMap* can be created with a no-arg constructor, or by passing in another map.

Suppose we need to represent the score for each person that played a game. If we assume that each person’s name is unique, then a map would be a good choice. Thus, we need a *map of scores where the key is a person’s name*. In other words, a map holds values, that is what it is storing, its objective; we just happen to use a key to access the values. Next, we illustrate some *Map* methods using the map of scores:

1. We define the map this way:

Map<String,Integer> hmScores = **new** HashMap<>();

1. The *put(key,value)* method is used to add a map entry to a map. For example:

hmScores.put("Felix", 42);

hmScores.put("Dania", 28);

hmScores.put("Cathy", 62);

If you *put(key,val)* with a key that exists, the value will be replaced in the existing entry with *val*. For example, the statement below will change the “Dania” entry so that the value is 99:

hmScores.put("Dania", 99);

1. The *size* method returns the number of entries in the map. For example:

**int** numEntries = hmScores.size();

1. The *get* method is used to retrieve a *value* given a *key.* For example:

**int** score = hmScores.get("Dania");

If the key does not exist, an exception is thrown. The *containsKey(key:String)* method is useful in this situation. For example:

**if**(hmScores.containsKey("Antonio")) {

score3 = hmScores.get("Antonio");

}

1. The *remove(key)* method removes the entry associated with the key, if it exists, and returns the value*.* If it doesn’t exist, an exception is thrown. For example:

score3 = hmScores.remove("Felix");

1. Internally, a map stores its keys in a *Set,* thus the keys in a map must be unique. The *keySet* method returns the *Set* of all the keys in a map. For example:

Set<String> names = hmScores.keySet();

returns the *Set* of all the names in the map. As shown in the class diagram above, the *keySet* method returns a *Set* object, not a *HashSet* nor other type of concrete set. Thus, we could not use a *HashSet* to hold the return from *keySet,* but you could use *Collection* since *Collection* is a super-interface for *Set.* The *Set* that the method returns is tied to the map, meaning changes in one will be reflected in the other. This implementation of *Set* does not support *add* or *addAll,* but supports the rest of the *Collection* methods. A call to *add* or *addAll* results in an exception being thrown. If you wanted a copy of the set that you could add to, you could create a *List* from the *Set*:

ArrayList<String> namesAL = **new** ArrayList<>(names);

namesAL.add("Lydia");

Or, create a *TreeSet* from the *Set*:

TreeSet<String> namesTS = **new** TreeSet<>(names);

namesTS.add("Eddy");

1. One way to iterate over all the entries in a map is:

**for**(String key : hmScores.keySet()) {

System.***out***.println("key=" + key + ", value=" + hmScores.get(key));

}

Similar to a *HashSet*, a *HashMap* stores its entries in no particular order.

1. Internally, a *Map* stores its values in a *Collection,* thus there can be duplicate values in a map as long as they have different keys. For example, two people could have the same score:

hmScores.put("Felix", 42);

hmScores.put("Dee", 42);

1. The *values* method returns a *Collection* of all the values in a map. For example:

Collection<Integer> scores = hmScores.values();

returns the *Collection* of all the scores in the map. The *Collection* that the method returns is tied to the map, meaning changes in one will be reflected in the other. This implementation does not support *add* or *addAll,* but supports the rest of the *Collection* methods. A call to *add* or *addAll* results in an exception being thrown. If you wanted a copy of the set that you could add to, you could create a *List* from the *Collection*:

ArrayList<Integer> scoresAL = **new** ArrayList<>(scores);

scoresAL.add(66);

Or, create a *TreeSet* from the *Collection* (losing any duplicates):

TreeSet<Integer> scoresTS = **new** TreeSet<>(scores);

scoresTS.add(62);

1. The *clear* method removes all entries from the map.

hmScores2.clear();

1. There are at least 4 ways to iterate over a map. The approach depends on what you are trying to do. The approaches are below.
2. Use case: need just the keys. Solution: get all the keys with the *keyset* method and iterate over them.

**for**(String key : hmScores.keySet()) {

System.***out***.println("key=" + key);

}

1. Use case: need just the values. Solution: Get all the values with the *values* method and iterate over them.

**for**(**int** value : hmScores.values()) {

System.***out***.println("value=" + value);

}

1. Use case: need both the keys and the values. Solution: get all the keys with the *keyset*, iterate over them, using each key to access the corresponding value with the *get* method.

**for**(String key : hmScores.keySet()) {

System.***out***.println("key=" + key + ", value=" + hmScores.get(key));

}

1. Use case: need both the keys and the values. Solution: get all the *entries* via the *entrySet* method. This method returns a *Set* of *Map.Entry* objects. A *Map.Entry* object contains a *key* and associated *value* accessed with a *getKey* and *getValue* method, respectively.

**for**(Map.Entry<String, Integer> entry : hmScores.entrySet()) {

System.***out***.println("key=" + entry.getKey() + ", value=" + entry.getValue());

}

1. The *containsValue* method returns *true* if *value* is found in the map; otherwise *false.* If the value is an object, then the corresponding class must override *equals* to work properly.

**boolean** isFound = hmScores.containsValue(42);

1. The overloaded constructor, HashMap(m:Map<? extends K, ? extends V>) creates a map from another map.

Map<String,Integer> hmScores2 = **new** HashMap<>(hmScores);

1. The *putAll* method accepts another map and adds all the entries into this map, replacing any that already exist.

hmScores2.putAll(hmScores);

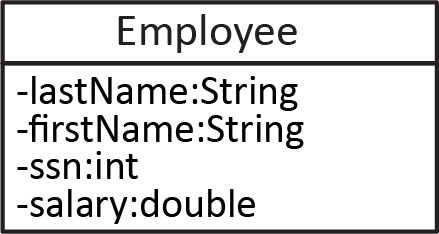
1. Finally, we can use custom objects as keys in a *HashMap;* however, it is subject to the same issues as storing custom objects in a *HashSet,* namely, you must override *hashCode* and *equals*.

## Exercises

1. (Solution in *exercise\_login\_account\_hashmap* package) Suppose you have (a) a *LoginAccount* class with the following fields (and associated getters): *userId, password, name, balance*. (b) a *LinkedList* of *LoginAccounts* named *accounts.* Write a snippet of code that creates a *HashMap* of *LoginAccount* objects using the *userID* (string) as the key, from this linked list (*accounts*).
2. (Solution in *exercise\_passwords\_hashmap* package) Suppose you have: (a) a *LinkedList* of userID’s (string) named *ids,* (b) a *LinkedList* of passwords (string) named *passwords,* (c) the two lists are the same size and have a 1-1 correspondence. In other words, the first userID in *ids* corresponds to the first password in *passwords*, *etc.* Write a snippet of code that creates a *HashMap* of the passwords where the key is the corresponding userID. However, only add the entry if the password has a length at least 6 and at least one digit. Also print the number of passwords that did not meet the criteria.
3. (Solution in *exercise\_add\_10\_to\_score\_hashmap* package) Suppose you have a *HashMap, mapScores* where the key is the *teamID* (integer) and the value is the team’s *score* (double). Write a snippet of code to add 10.0 to each team’s score for the teams with a *teamID* of 5 or more.
4. (Solution in *exercise\_merge\_scores\_hashmap* package) Suppose you have two *HashMaps, mapScores1* and *mapScores2*. Each map has a key which is the *teamID* (integer) and the value which is the team’s *score* (double). Write a snippet of code to add any entries from *mapScores1* that are not in *mapScores2* to *mapScores2.* If a team in *mapScores1* is in *mapScores2* then modify the value in *mapScores2* so that it is the sum of the two scores. For example:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input |  | Result |  | Notes |
| |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | | |  |  | | --- | --- | | *mapScores1* | | | *Key* | *Value* | | 1 | 100.0 | | 2 | 500.0 | | 3 | 400.0 | | 4 | 200.0 | | 5 | 900.0 | | |  |  | | --- | --- | | *mapScores2* | | | *Key* | *Value* | | 3 | 200.0 | | 4 | 700.0 | | 8 | 100.0 | | 10 | 400.0 | | |  | |  |  | | --- | --- | | *mapScores2* | | | *Key* | *Value* | | 1 | 100.0 | | 2 | 500.0 | | 3 | 600.0 | | 4 | 900.0 | | 5 | 900.0 | | 8 | 100.0 | | 10 | 400.0 | |  | * Since the key, 1 in *mapscores1* does not occur in *mapscores2,* it is added. * Since the key, 3 in *mapscores1* also occurs in *mapscores2,* then the values for the two maps are added, replacing the value in *mapscores2* |

# Example 1: Map of Employee Objects

The example in this section is in the *example\_ssn\_employee* package.

Consider the *Employee* class on the right. The example below will utilize a *map of Employee objects where the employee’s SSN is the key.* In this case, the key is a property of the *Employee* class. We see how that comes into play, below, when we put an employee into the map.

// Create map

Map<Integer,Employee> hmEmployees = **new** HashMap< >();

// Create employees

Employee e1 = **new** Employee("Lyton", "Xavier", 243558673, 77.88);

...

// Put employee in map

hmEmployees.put(e1.getSSNum(), e1);

...

// Get all the SSN’s

Set<Integer> keys = hmEmployees.keySet();

// Iterate over all the employees via the SSN’s

**for**(**int** key : keys) {

Employee e = hmEmployees.get(key);

}

// Get all the employees

Collection<Employee> emps = hmEmployees.values();

// Iterate over the employees

**for**(Employee e : hmEmployees.values()) {

System.***out***.println(e);

}

# Example 2: Map of Lists

The example in this section is in the *example\_map\_of\_lists* package.

Suppose we need to represent a person’s favorite musical artists and we need to do this for a bunch of people. For example, something like this:

name=Dave, fav music: Bob Dylan, Dead, John Prine,

name=Lee, fav music: Dead, Bob Dylan, Steve Miller, Digit 60,

name=Anna, fav music: Boz Scaggs, Dead,

One way to do this would be to use a map of lists, where the key is a person’s name (assuming unique).

We create the map below. Note the generic type for the *value* highlighted in yellow below.

HashMap<String,List<String>> hmFavMusic = **new** HashMap<>();

To use this map, we can create a list of artists:

List<String> artists1 = **new** ArrayList<>();

artists1.add("Bob Dylan");

artists1.add("Dead");

artists1.add("John Prine");

Then, we can add the list to the map using a name as a key:

hmFavMusic.put("Dave", artists1);

Finally, one way we could iterate over all the lists is:

**for**(String key : hmFavMusic.keySet()) {

List<String> artists = hmFavMusic.get(key);

**for**(String artist : artists) {

System.***out***.printf(artist + ", ");

}

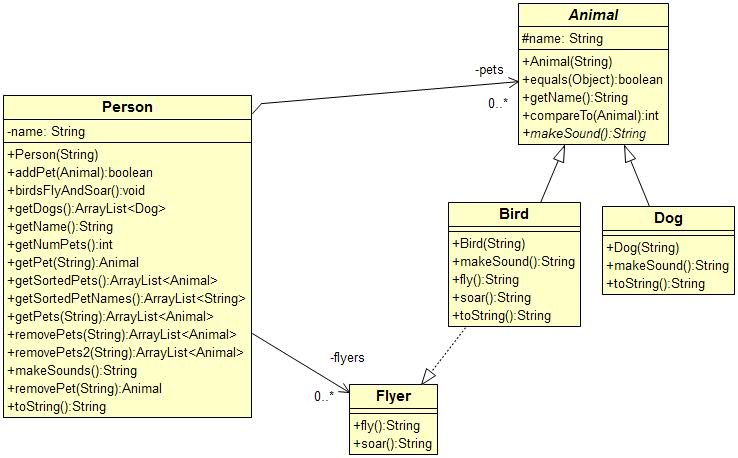
System.***out***.println();

}

# Example 3: 1 to Many

The code for this example is in the *example\_person\_animals\_ver4* package.

Consider the example shown in the class diagram below. We considered this example extensively in Ch. 5. A *Person* has many *Animals* and also has many *Flyers*. Here, we replace the use of *ArrayList*s to manage the two 1-many relationships with *Maps*.

****

1. Associations – Previously the two 1-many relationships were implemented as *ArrayList*:

**private** ArrayList<Animal> pets = **new** ArrayList<>();

**private** ArrayList<Flyer> flyers = **new** ArrayList<>();

Here, we replace the *ArrayLists* with *HashMaps*.

**private** HashMap<String,Animal> pets = **new** HashMap<>();

**private** HashMap<String,Flyer> flyers = **new** HashMap<>();

using the *Animal’s name* as the key and the value is the corresponding *Animal* object for the *pets* map or *Flyer* for the *flyers* map.

1. The *addPet* method:

|  |  |
| --- | --- |
| ***HashMap* Implementation** | ***ArrayList* Implementation (Ch 5)** |
| **public** **boolean** addPet(Animal a) {  String name = a.getName();  **if**(!pets.containsKey(name)) {  pets.put(name,a);  **if**(a **instanceof** Flyer) {  Flyer f = (Flyer)a;  flyers.put(name,f);  }  **return** **true**;  }  **return** **false**;  } | **public** **boolean** addPet(Animal a) {  **if**(!pets.contains(a)) {  pets.add(a);  **if**(a **instanceof** Flyer) {  Flyer f = (Flyer)a;  flyers.add(f);  }  **return** **true**;  }  **return** **false**;  } |

1. The *getNumPets* method does not change as a map has a *size* method just as a list does:

**public** **int** getNumPets() {

**return** pets.size();

}

1. In Ch 5, we had two *getPet* methods:
2. *getPet(index:int):Animal* – Returns the animal at *index*. With a map, this method no longer makes sense as there is no index to access animals.
3. *getPet(name:String):Animal* – Returns the animal with *name*. There, we overrode *equals* in the *Animal* class so that two *Animals* were equal if they have the same name which allowed us to use *indexOf* to find the location of the animal as shown in the code below on the left. Note that that method found the index and then called *getPet(index:int)* to complete the retrieval.

|  |  |
| --- | --- |
| ***getPet(name:String)* – Ch 5** | ***getPet(index:int)* – Ch 5** |
| **public** Animal getPet(String name) {  Animal a = **new** Dog(name);  **int** loc = pets.indexOf(a);  **return** pets.get(loc);  } | **public** Animal getPet(**int** i) {  **if**( i>=0 && i<pets.size()) {  **return** pets.get(i);  }  **return** **null**;  } |

To implement *getPet(name:String):Animal* with a map,

**public** Animal getPet(String name) {

**if**(pets.containsKey(name)) {

**return** pets.get(name);

}

**return** **null**;

}

1. The situation with *removePet* is similar to *getPet*. We no longer have a *removePet(index:int)* method. The *removePet(name:String)* method with the map implementation is shown below:

**public** Animal removePet(String name) {

**if**(pets.containsKey(name)) {

Animal a = pets.get(name);

pets.remove(name);

**if**(a **instanceof** Flyer) {

flyers.remove(name);

}

**return** a;

}

**return** **null**;

}

1. The *getSortedPets* method changes slightly:

|  |  |
| --- | --- |
| ***HashMap* Implementation** | ***ArrayList* Implementation (Ch 5)** |
| **public** ArrayList<Animal> getSortedPets() {  ArrayList<Animal> sorted =  **new** ArrayList<>(pets.values());  Collections.*sort*(sorted);  **return** sorted;  } | **public** ArrayList<Animal> getSortedPets() {  ArrayList<Animal> sorted =  **new** ArrayList<>(pets);  Collections.*sort*(sorted);  **return** sorted;  } |

1. We add a *getSortedPetNames* method that returns a list of the pet names, sorted.

**public** ArrayList<String> getSortedPetNames() {

ArrayList<String> sorted = **new** ArrayList<>(pets.keySet());

Collections.*sort*(sorted);

**return** sorted;

}

1. The *birdsFlyAndSoar* and *getDogs* methods change slightly (*makeSounds* is similar):

|  |  |
| --- | --- |
| ***HashMap* Implementation** | ***ArrayList* Implementation (Ch 5)** |
| **public** **void** birdsFlyAndSoar() {  **for**(Flyer f : flyers.values()) {  f.fly();  f.soar();  }  }  **public** ArrayList<Dog> getDogs() {  ArrayList<Dog> dogs =  **new** ArrayList<>();  **for**(Animal a : pets.values()) {  **if**(a **instanceof** Dog) {  dogs.add((Dog)a);  }  }  **return** dogs;  } | **public** **void** birdsFlyAndSoar() {  **for**(Flyer f : flyers) {  f.fly();  f.soar();  }  }  **public** ArrayList<Dog> getDogs() {  ArrayList<Dog> dogs =  **new** ArrayList<>();  **for**(Animal a : pets) {  **if**(a **instanceof** Dog) {  dogs.add((Dog)a);  }  }  **return** dogs;  } |

1. We add a *getPets* method that accepts a string representing the beginning characters of a name, a *partialKey*. The method returns a list of *Animals* whose name begins with *partialKey*:

**public** ArrayList<Animal> getPets(String partialKey) {

ArrayList<Animal> matches = **new** ArrayList<>();

**int** len = partialKey.length();

**for**(String name : pets.keySet()) {

**if**(name.length()<partialKey.length()) {

**continue**;

}

String beginningOfName = name.substring(0, len);

**if**(beginningOfName.equals(partialKey)) {

matches.add(pets.get(name));

}

}

**return** matches;

}

1. We add a *removePets* method that accepts a string representing the beginning characters of a name, a *partialKey*. The method returns and removes all *Animals* from the map whose name begins with *partialKey*. Note that this method uses *getPets* to get the *Animal* objects to remove. Then, the remove is easy.

**public** ArrayList<Animal> removePets(String partialKey) {

// Use getPets method to get a list of all pets that match

// the partialKey. Thus, these are the pets that will be removed.

ArrayList<Animal> removed = getPets(partialKey);

**for**(Animal a : removed) {

pets.remove(a.getName());

**if**(a **instanceof** Flyer) {

flyers.remove(a.getName());

}

}

**return** removed;

}

# The *LinkedHashMap* Class

The *LinkedHashMap* class is identical to *HashMap* except that the order of insertion is preserved.

(Optional) The entries in a *LinkedHashMap* can also be accessed in the order in which they were last accessed, from least recently accessed to most recently. This is called *access order* and is specified, for example, by using *true* for the last argument in the constructor below:

LinkedHashMap(initialCapacity, loadFactor, true).

This can be used in what is called a *Least Recently Used Cache[[2]](#footnote-2)* (LRU) where you want to maintain a finite sized cache and when a new entry is added which increases the size beyond the desired maximum, the *stalest* (least recently used) entry is removed. It supports a protected method, *removeEldestEntry* which can be overridden when extending *LinkedHashMap*.

# The *TreeMap* Class

A *TreeMap* is a simply a map where the keys are ordered (where *Comparable* or *Comparator* is used for the ordering).Below, we create a *TreeMap* of *Employee* objects where the keys are ordered.

// Create map

TreeMap<Integer,Employee> tmEmployees = **new** TreeMap<>();

// Create employees

Employee e1 = **new** Employee("Lyton", "Xavier", 243558673, 77.88);

...

// Put employee in map

tmEmployees.put(e1.getSSNum(), e1);

...

A *TreeMap* also supports a number of additional methods via the *SortedMap* and *NavigableMap* interfaces. More information about this is found in an [appendix](#Appendix_SortedMap_NavigableMap).

## Exercises

For the next four problems: Suppose you have a *LoginAccount* class (same as a previous exercise) with the following string fields (and associated getters): *userId, password, name* and *balance* (double). Solutions for the following four problems are in the *exercise\_loginaccount\_treemap* package, *LoginAccountTest* class*.*

1. (*createTreeMapFromHashMap* method) Suppose you have a *HashMap* of *LoginAccounts* named *hmAccounts* where *userId* is the key. Write a snippet of code to create a *TreeMap* named *tmAccounts* which is the same as *hmAccounts* except the keys are ordered. Hint: you can do this in one line with a constructor (preferred), or you can use a loop.
2. (*getPasswords* method) Suppose you have a *TreeMap* of *LoginAccounts* where *userId* is the key. Write a method, *getPasswords* that accepts such a *TreeMap* of *LoginAccounts* and returns a list of just the passwords.
3. (*getPasswords2* method) Suppose that you have a *TreeMap* of *LoginAccounts* where *userId* is the key. Write a method, *getPasswords2* that accepts such a *TreeMap* of *LoginAccounts* and returns a set of the unique passwords (*i.e.* just one occurrence of each password).
4. (*removeAccounts* method) Suppose that you have a *TreeMap* of *LoginAccounts* where *userId* is the key. Write a method, *removeAccounts* that accepts such a *TreeMap* of *LoginAccounts* and removes accounts where the userID doesn’t contain the “@” symbol. This method does not return anything. Hint: this requires[[3]](#footnote-3) two loops, one to collect the *userId*s that don’t contain “@”, and another to loop over this list and remove each from the map.
5. (Solution in *exercise\_swap\_key\_value* package) Consider a *TreeMap* where the key is a character and the value is an integer. For example, see the *Input* table (map) below on the left. Write a static method, *swapKeyValue* that accepts such a map. Next, consider a *TreeMap* where the key is an integer and the value is a string. For example, see the *Output* table (map) below in the middle. The method should swap the key and value from the input map such that if a value occurs more than once in the input map, then the corresponding keys are concatenated to form the value for the output map. For example:

|  |  |  |  |
| --- | --- | --- | --- |
| Input |  | Output |  |
| |  |  | | --- | --- | | *Key* | *Value* | | A | 8 | | B | 2 | | C | 4 | | G | 2 | | L | 8 | | P | 2 | | R | 1 | | V | 3 | |  | |  |  | | --- | --- | | *Key* | *Value* | | 1 | R | | 2 | BGP | | 3 | V | | 4 | C | | 8 | AL | | Note: in the input map that the value:   * 1 occurs only once, so the output map has a key of 1 and the value “R”. * 2 occurs three times, so the output map has a key of 2 and the value “BGP” * 8 occurs twice. Thus, the output map has a key of 8 and the value is “AL”. |

# Example 4: Occurrences of Words & Modelling

In this section, we will look at an example that uses a map, and then think about how to develop it as part of a larger system.

## Problem Description

Consider a scenario where we have an array of keywords and we want to read a text file and count the number of occurrences of each keyword. For example, if the keywords are:

String[] *happyWordsAry* = {"content", "cheerful", "cheery",

"merry", "joyful", "joy", "jovial", "jolly", "joking", "joke", "jocular", "gleeful",

"glee", "carefree", "untroubled", "delighted", "delight", "smiling", "smile",

"delighted", "delight", "elated", "elation",

"glad", "joyous", "jubilant", "lively", "pleased", "thrilled", "happy",

"upbeat", "blessed", "blest", "blissful", "chipper", "chirpy", "content",

"convivial", "gay", "gratified", "laughing", "laugh", "mirthful", "peppy",

"playful"};

And we compare these to the text of [*The Catcher in the Rye*](https://en.wikipedia.org/wiki/The_Catcher_in_the_Rye), the result is (only the keywords that actually occurred in the text are shown):

|  |  |
| --- | --- |
| Keyword | Count |
| gay | 1 |
| glad | 14 |
| happy | 7 |
| joy | 1 |
| laugh | 8 |
| laughing | 4 |
| playful | 3 |
| smile | 5 |
| smiling | 1 |

## Algorithm

A *TreeMap* (or other type of map) will be useful to store the results. We can use a keyword as the key, and the number of occurrences as the value. For example:

TreeMap<String,Integer> tmOccurrences = **new** TreeMap<>();

Suppose the keywords are in a set:

Set<String> *keywords* = **new** HashSet<>(Arrays.*asList*(*“happy”, “joy”, “upbeat”, ...*));

Finally, we could loop over each word (token) in the file and then use the algorithm below to add the word to the map, or increase the count if it is already there:

For each *word* in file

If *word* is a *keyword*

If *returnMap* doesn’t already have the *word*

Put *word* in *returnMap* with a count of 1

Else *returnMap* has the *word*

*newCount* = *currentCount* + 1

Put word in *returnMap* with a count of *newCount*

## Method

Next, we write a method that accepts a *File* and a *Set* of keywords and returns a *Map* of occurrences of keywords. The solution is found in the code download in the *example\_count\_word\_occurrences\_map* package.

**public** **static** TreeMap<String,Integer> getKeywordOccurrences(File file, Set<String> keyWords) **throws** FileNotFoundException {

TreeMap<String,Integer> tmOccurrences = **new** TreeMap<>();

Scanner input = **new** Scanner(file);

**while** (input.hasNext()) {

String word = input.next(); // get next word

word = *getCleanWord*(word); // clean word up

**if** (keyWords.contains(word)) { // word is a keyword

**if** (!tmOccurrences.containsKey(word)) { // word not in map

tmOccurrences.put(word, 1);

}

**else** { // word in map

**int** count = tmOccurrences.get(word); // get current count

tmOccurrences.put(word, ++count); // update count

}

}

}

input.close();

**return** tmOccurrences;

}

Note, that we use *getCleanWord* aboveto remove any leading or trailing blank spaces, convert the word to lower-case, and remove any punctuation or double quotes.

As a variation, we could write another method to count all the occurrences of each word in the file by simply eliminating this *if* statement below (in the code above):

**if** (keyWords.contains(word)) {

This is also contained in the code download, the method, *getWordOccurrences*.

## Single Responsibility Principle

Next, we consider how to take the methods from the previous section and make them more reusable. But first, we consider a design principle that will help guide us. There are a number of OO design principles that help make software designs more understandable, reusable, and maintainable. Here, we discuss just one, a very import one, and one I think you should seek to incorporate completely into the way you think about design. The Single Responsibility Principle:

*A class should have only one reason to change, in other words, a class should have only one responsibility*

As we develop a software system, classes can become bloated; they take on too much responsibility. If we look carefully at such classes, we can often break them into two (or more) classes each with a single responsibility.

Here is a simple example that violates the single responsibility principle: Suppose we have an Employee class with name, payrate, hours worked, *etc* and a method to calculate their pay. Suppose this class also has a method to write the instance variables to a text file (or database, *etc*.). This is two responsibilities: one is to manage the employee’s attributes and the other is to persist it to disk. We always have a separate class(es) to do data persistence.

## Refactoring for Reuse

The code below is in the code download in a package named: *example\_text\_analyzer*.

First, we need to pull the reading out into its own class. Thus, we can provide a *TextReader* class with a *getWords* static method that accepts a *File* and returns a *List<String>* of the words in the file as shown below. We use a *static* method because this is a utility method that doesn’t rely on any state information. It only needs what is passed into it, the *File*.

**public** **class** TextReader {

**public** **static** List<String> getWords(File file) {

List<String> words = **new** ArrayList<>();

Scanner input;

**try** {

input = **new** Scanner(file);

**while**(input.hasNext()) {

words.add(input.next());

}

input.close();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

**return** words;

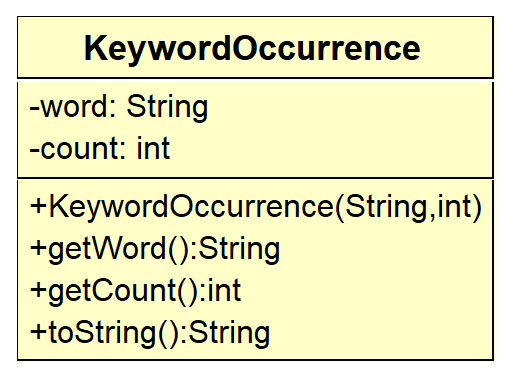
}

}

Next, we need to change the earlier *getKeywordOccurrences* method so that it doesn’t read the words, but instead accepts a list of words as input.

**static** TreeMap<String,Integer> getKeywordOccurrences(List<String> words, Set<String> keywords) {

Currently, the method returns a *Map* of the counts of each keyword. Suppose, in addition, we also need to return the total number of unique keywords found, and the total number of keywords. In the example in [Section 8.1](#_Problem_Description), there were 9 different keywords found (the number of keys in the map), and 44 instances of those (the sum of all the values in the map). This is a frequent problem we encounter – needing to return more than one thing. Java only allows us to return one object – but that is actually a good thing if we are trying to follow the single responsibility principle. We create a class to hold the items we need to return. First, we define a *KeywordOccurrence* class to record the word and the count:

**public** **class** KeywordOccurrence {

**private** String word;

**private** **int** count;

**public** KeywordOccurrence(String word, **int** count) {

**this**.word = word;

**this**.count = count;

}

**public** String getWord() {

**return** word;

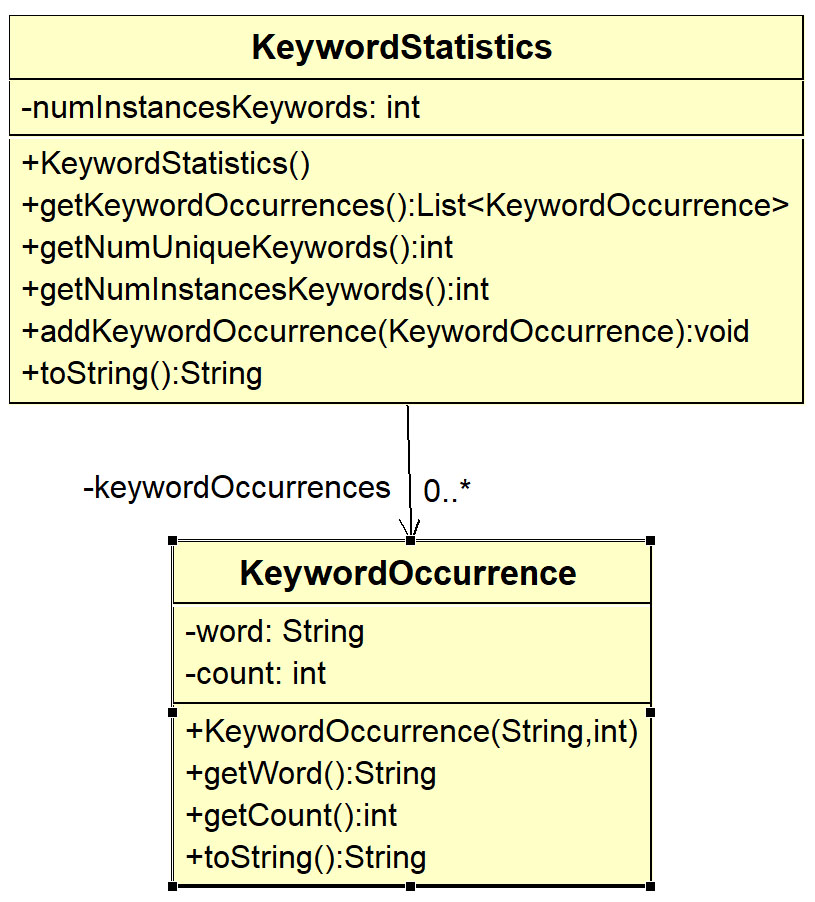
}

**public** **int** getCount() {

**return** count;

}

...}

Next, we define a *KeywordStatistics* class that encapsulates the list of *KeywordOccurrences* and the other two statistics.

**public** **class** KeywordStatistics {

**private** List<KeywordOccurrence> keywordOccurrences = **new** ArrayList<>();

**private** **int** numInstancesKeywords = 0;

**public** KeywordStatistics() {}

**public** List<KeywordOccurrence> getKeywordOccurrences() {

**return** keywordOccurrences;

}

**public** **int** getNumUniqueKeywords() {

**return** keywordOccurrences.size();

}

**public** **int** getNumInstancesKeywords() {

**return** numInstancesKeywords;

}

**public** **void** addKeywordOccurrence(KeywordOccurrence keywordOccurrence) {

numInstancesKeywords += keywordOccurrence.getCount();

keywordOccurrences.add(keywordOccurrence);

}

...

}

Finally, we can change the return type of *getKeywordOccurrences* to *KeywordStatistics* and add code to build the *KeywordStatistics* object.

**static** KeywordStatistics getKeywordOccurrences(List<String> words, Set<String> keywords) {

TreeMap<String,Integer> tmOccurrences = **new** TreeMap<>();

**for**(String word : words) {

word = *getCleanWord*(word);

**if** (keywords.contains(word)) {

**if** (!tmOccurrences.containsKey(word)) {

tmOccurrences.put(word, 1);

}

**else** {

**int** count = tmOccurrences.get(word);

tmOccurrences.put(word, ++count);

}

}

}

**return** *buildKeywordStatistics*(tmOccurrences);

}

We use a helper method, *buildKeywordStatistics* that transfers the *Map* of results into a *KeywordStatistics* object. Sometimes, we call (loosely) such a method, a *factory method*. The method is a factory for making a *KeywordStatistics* object.

**static** KeywordStatistics buildKeywordStatistics(TreeMap<String,Integer> tmOccurrences) {

KeywordStatistics stats = **new** KeywordStatistics();

**for**(String word : tmOccurrences.keySet()) {

**int** count = tmOccurrences.get(word);

KeywordOccurrence occurrence = **new** KeywordOccurrence(word, count);

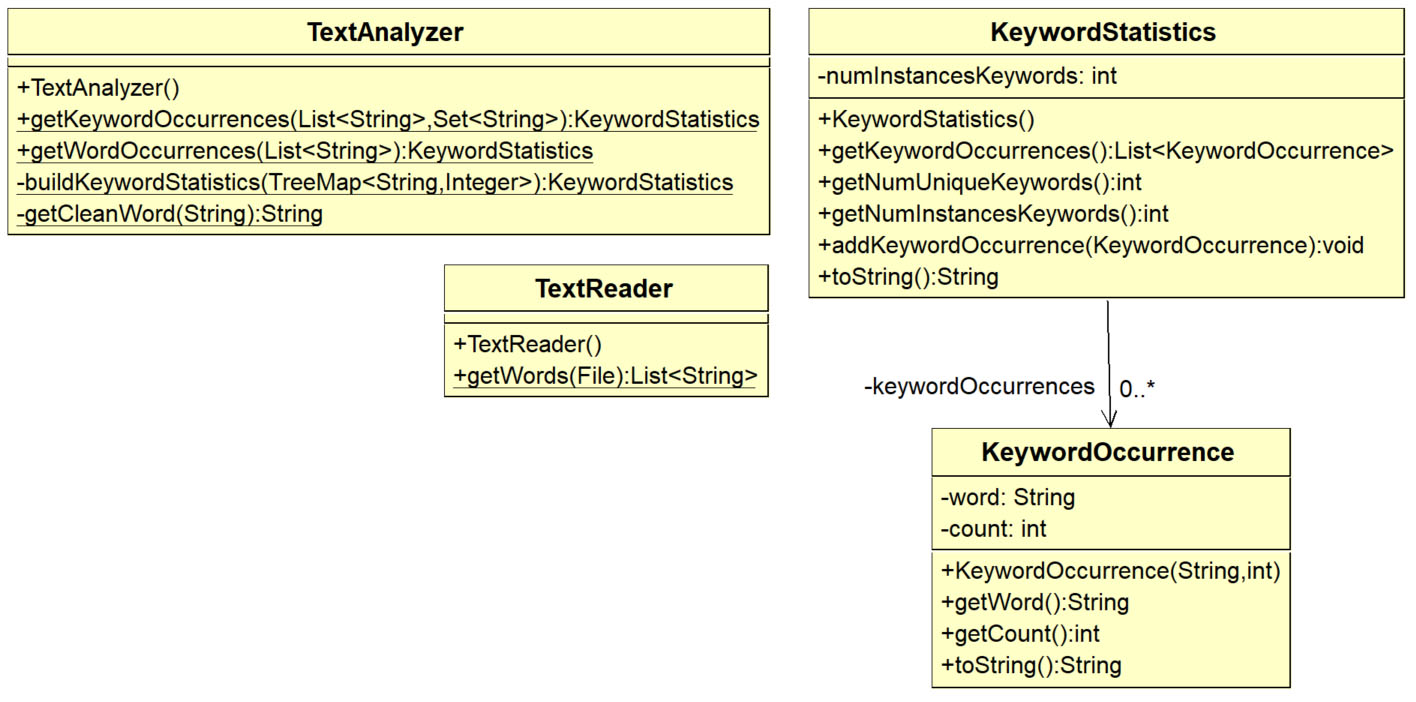
stats.addKeywordOccurrence(occurrence);

}

**return** stats;

}

The final issue is where does this method belong? What class? We choose to write a class, *TextAnalyzer,* in which we place the static method, *getKeywordOccurrences*. The class diagram for these classes is shown below.



Test code would look like this:

File file = **new** File(*filename*);

List<String> words = TextReader.*getWords*(file);

KeywordStatistics stats = TextAnalyzer.*getKeywordOccurrences*(words, *happyWords*);

System.***out***.println(stats);

Where *filename* and *happyWords* are defined as constants:

**static** **final** Set<String> ***happyWords*** = **new** HashSet<>(Arrays.*asList*("glad", "playful", "joy"));

**static** **final** String ***filename*** = "src/example\_text\_analyzer/Catcher\_in\_the\_Rye.txt";

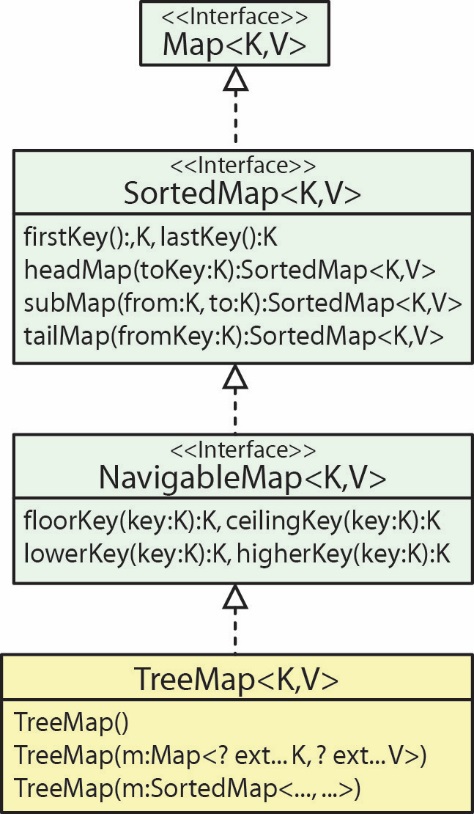
## Exercises

1. (Solution NOT available at present, eventually will be in *example\_text\_analzer* package) In the *KeywordStatistics* class above, the *getKeywordOccurrences* method returns a list of *KeywordOccurrence* objects that is simply ordered on the *word.* An example is shown in left column of the table below. Add a method to this class, *getSortedKeywordOccurrences* that returns a list that is ordered on *count* (number of occurrences), and if there is a tie, then ordered on the *word*. An example is shown in the right column in the table below. Hints: (a) define a *Comparator* to order *KeywordOccurrence* objects. (b) create a copy of the *keywordOccurrences,* (c) sort the copy using the comparator, (d) return this list.

|  |  |
| --- | --- |
| ***getKeywordOccurrences*** | ***getSortedKeywordOccurrences*** |
| gay-1 | gay-1 |
| glad-14 | joy-1 |
| happy-7 | smiling-1 |
| joy-1 | playful-3 |
| laugh-8 | laughing-4 |
| laughing-4 | smile-5 |
| playful-3 | happy-7 |
| smile-5 | laugh-8 |
| smiling-1 | glad-14 |

Appendix

1. The *SortedMap* & *NavigableMap* Interfaces

Some sample code if in the *example\_sorted\_map* package. ****As shown in the class diagram above on the right, a *TreeMap* implements methods from *SortedMap* that are analogous to the methods *TreeSet* inherits from *SortedSet:*

|  |  |
| --- | --- |
| **TreeMap (SortedMap)** | **TreeSet (SortedSet)** |
| firstKey():K | first():E |
| lastKey():K | last():E |
| headMap(toKey:K)  :SortedMap<K,V> | headSet(toElement:E)  :SortedSet<E> |
| tailMap(fromKey:K)  :SortedMap<K,V> | tailSet(fromElement:E)  :SortedSet<E> |
| subMap(from:K,to:K)  :SortedMap<K,V> | subSet(from:E, to:E)  :SortedSet<E> |

For example, *headMap(toKey)* returns a *SortedMap* of map entries corresponding to the keys that are strictly less than *toKey,* {x|x<toKey}.

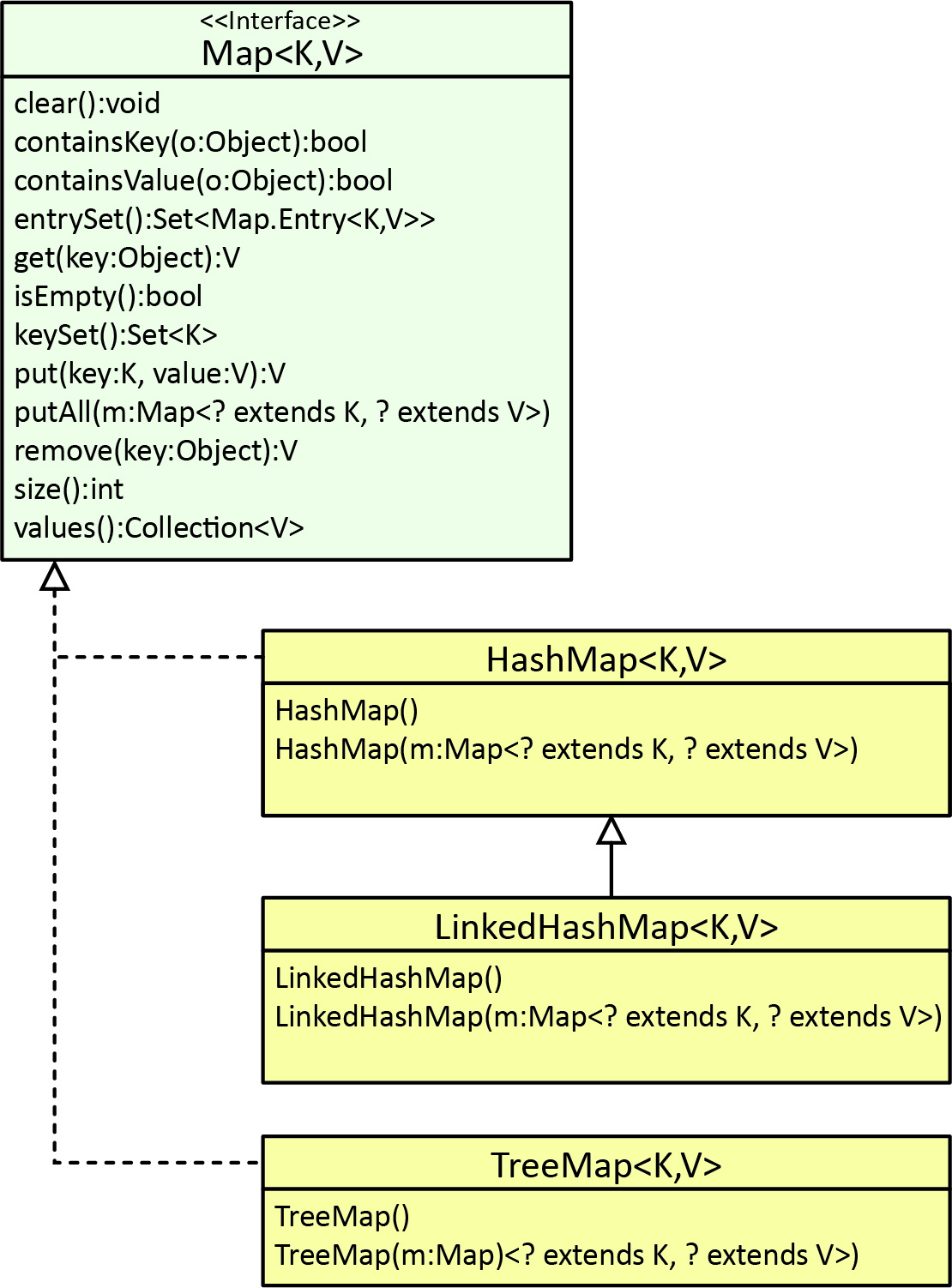
Also shown in the class diagram above on the right, a *TreeMap* implements methods from *NavigableMap* that are analogous to the methods *TreeSet* inherits from *NavigableSet:*

|  |  |
| --- | --- |
| **TreeMap (NavigableMap)** | **TreeSet (NavigableSet)** |
| floorKey(key:K):K | floor(e:E):E |
| lowerKey(key:K):K | lower(e:E):E |
| ceilingKey(key:K):K | ceiling(e:E) :E |
| higherKey(key:K):K | higher(e:E):E |

For example, *floorKey(key)* returns the greatest key less than or equal to the given *key*, or null if there is no such key.

1. Class Diagram for Map Classes

These are the classes and methods you are responsible for on a test. I will provide a copy of this image on a test.



1. <https://docs.oracle.com/javase/tutorial/collections/TOC.html> [↑](#footnote-ref-1)
2. <https://en.wikipedia.org/wiki/Cache_replacement_policies> [↑](#footnote-ref-2)
3. Unless you an iterator over the *entrySet()* [↑](#footnote-ref-3)