Chapter 8 – Lists & Sets

Contents

[1 Collections 2](#_Toc161753997)

[2 The *List* Interface & Additional *ArrayList* Methods. 3](#_Toc161753998)

[2.1 Exercises 4](#_Toc161753999)

[3 Iterators 5](#_Toc161754000)

[3.1 Iterating over a Collection with an Iterator 5](#_Toc161754001)

[3.2 Filtering a Collection with an Iterator – Removing Elements 5](#_Toc161754002)

[3.3 Filtering a Collection with an Iterator – Removing & Returning Elements 6](#_Toc161754003)

[3.4 Exercises 7](#_Toc161754004)

[4 The *LinkedList* Class 7](#_Toc161754005)

[5 The *Comparator* Interface 9](#_Toc161754006)

[5.1 Exercises 12](#_Toc161754007)

[6 The *Set* Interface 13](#_Toc161754008)

[7 The *HashSet* Class 14](#_Toc161754009)

[7.1 Exercises 15](#_Toc161754010)

[8 The *LinkedHashSet* Class 15](#_Toc161754011)

[9 The *TreeSet* Class 15](#_Toc161754012)

[9.1 Exercises 16](#_Toc161754013)

[10 The *TreeSet* Class with Custom Objects 16](#_Toc161754014)

[10.1 Exercises 18](#_Toc161754015)

[11 Performance of Sets & Lists 19](#_Toc161754016)

[12 Example: Counting Keywords 20](#_Toc161754017)

[Appendix 1 Filtering a Collection Incorrectly 22](#_Toc161754018)

[Appendix 1.1 Filtering a Collection with a *for-each* Loop Incorrectly 22](#_Toc161754019)

[Appendix 1.2 Filtering a Collection with a Forward, Indexed Loop Incorrectly 22](#_Toc161754020)

[Appendix 2 Other Approaches to Filtering a Collection 23](#_Toc161754021)

[Appendix 3 Static Methods for Lists & Collections 24](#_Toc161754022)

[Appendix 4 Class Diagram for Collection Classes 25](#_Toc161754023)

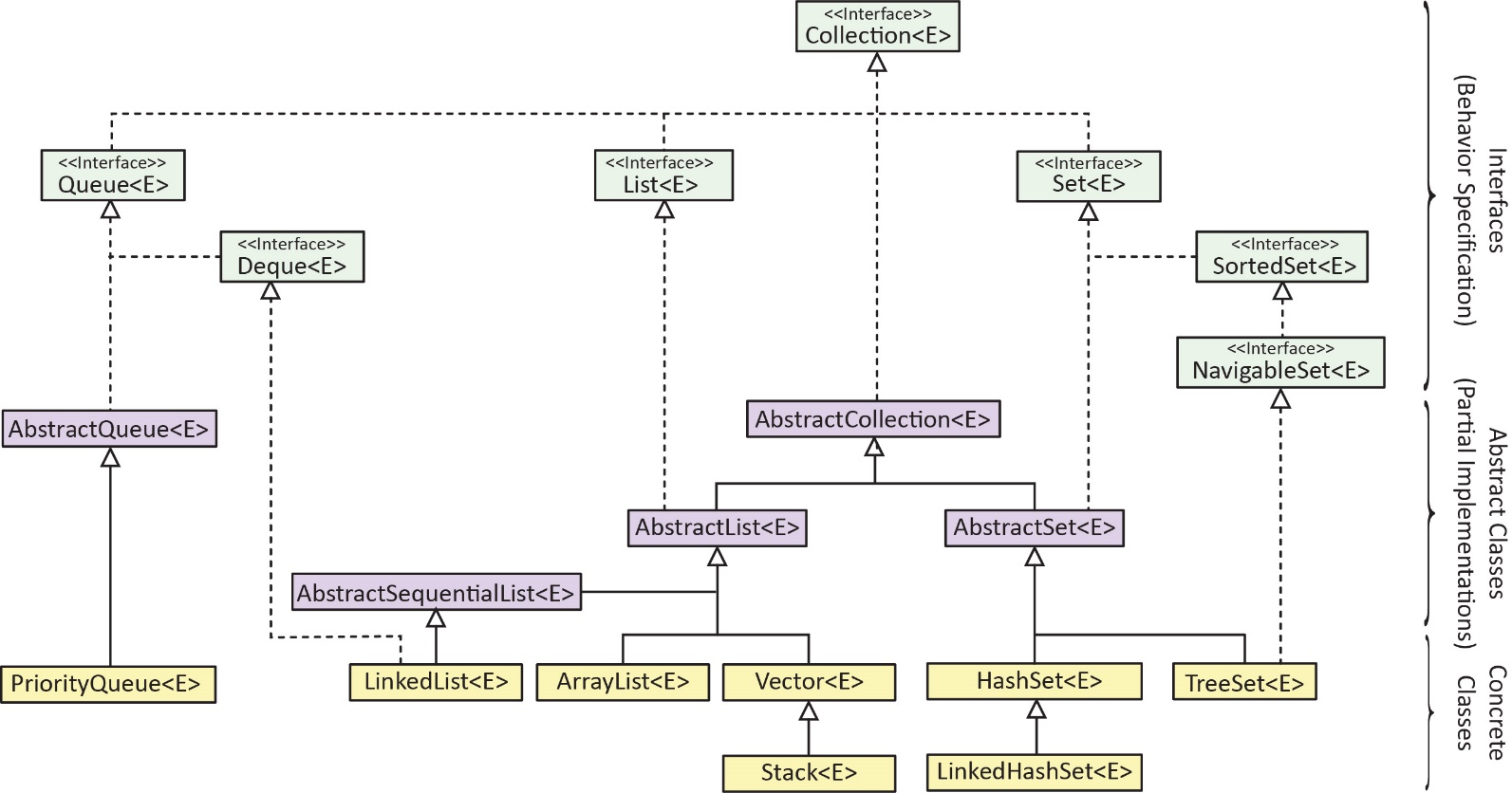
[Appendix 5 The Vector & Stack Classes 27](#_Toc161754024)

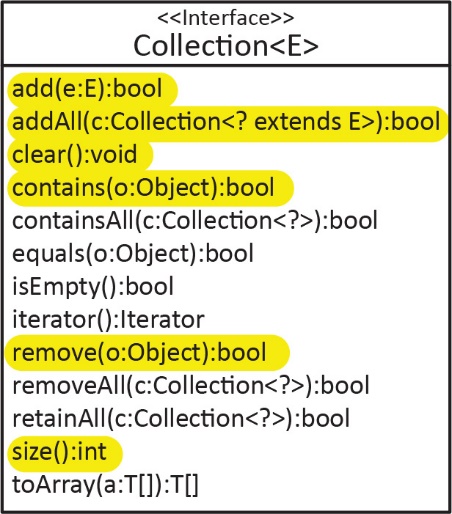
[Appendix 6 Queues and Priority Queues 28](#_Toc161754025)

[Appendix 7 The *SortedSet* & *NavigableSet* Interfaces 30](#_Toc161754026)

# Collections

A *collection class* (also called a *container class*) is a generic term used to describe a class that is used to hold a group of elements. For example, an *ArrayList* is a collection class. The Java Collections Framework (JCF) defines the organization of all the collection classes in Java. It is composed of classes, interfaces, and abstract classes as (partially) shown in the class diagram below. In the JCF, interfaces (green boxes) are the top layer and define the behaviors, abstract classes (purple boxes) are the middle layer that provide partial implementations of the behaviors, and at the lower level, concrete classes (yellow boxes) provide any further implementation that is necessary. There, the green rectangles are interfaces, the purple rectangles are abstract classes, and the yellow rectangles are concrete classes.



Some of the *Collection* interface methods are shown in the diagram on the right. The highlighted methods are ones we have considered previously in the *ArrayList* class. All collections have these methods. Notice that this interface does not define, a get(i:int) method. This is defined in the *List* sub-interface which we consider shortly.

*Collections* are further broken down into sub-interfaces representing *Lists, Queues,* and *Sets*.

|  |  |
| --- | --- |
| **Interface** | **Description** |
| *List* | An ordered collection where elements can be accessed by an index |
| *Set* | A collection that does not allow duplicate elements, nor can elements be accessed by an index |
| *Queue* | A collection that is designed to hold elements that wait to be processed |

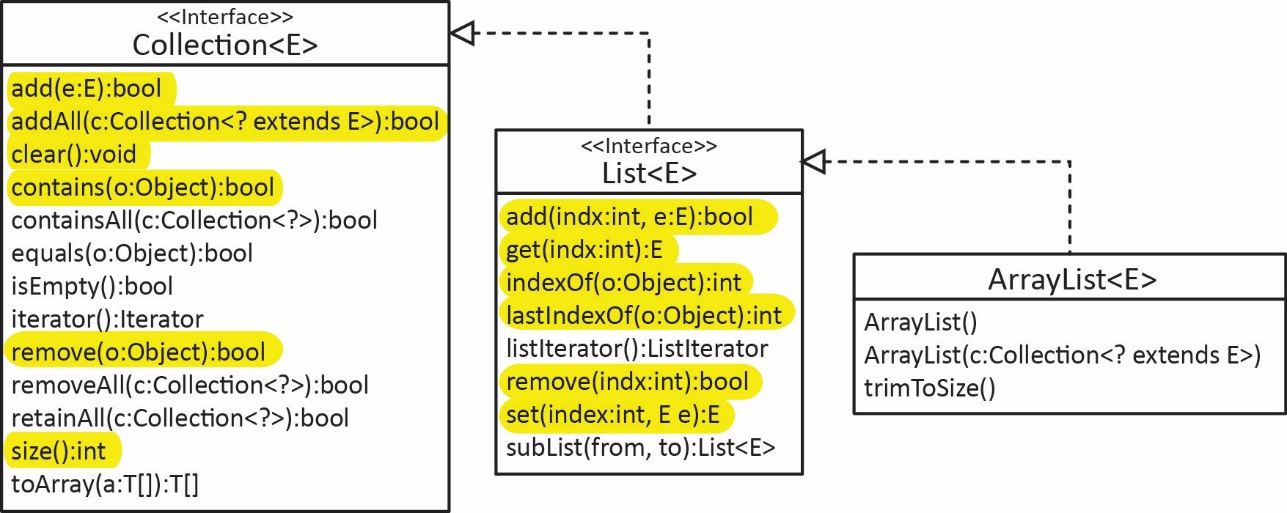
In this chapter, we will consider the (a) *List* interface and the *ArrayList* and *LinkedList* classes, (b) *Set* interface and *HashSet, LinkedHashSet,* and *TreeSet* classes. The Appendices consider the: (a) the [*Vector* and *Stack*](#Appendix_vector_stack)classes, (b) the [*Queue*](#Appendix_queue)interface and the *PriorityQueue* class. A detailed tutorial about the JCF is found at:

<http://docs.oracle.com/javase/tutorial/collections/>

# The *List* Interface & Additional *ArrayList* Methods.

The code for the example in this section is in the *example\_collection\_methods* package.

The class diagram below shows that all the methods for the *ArrayList* class that we have studied so far (highlighted) have been defined in the *Collection* and *List* interfaces. The *List* interface specifies that any implementing class must be indexed. Thus, it provides methods that require an index to reference an object. For example: add(i:int), get(i:int), remove(i:int), set(i:int,val).



As we saw in a previous chapter, the *ArrayList* has an *addAll* method that accepts another *ArrayList* and adds those elements to the list. As shown above, the *Collection* interface specifies that *any* type of *Collection* (of the same type or subtype) can be used as an argument. As a review:

List<String> cities = **new** ArrayList<>(Arrays.*asList*("New York", "San Francisco", "Moab"));

List<String> cities2 = **new** ArrayList<>(Arrays.*asList*("Atlanta", "Memphis"));

cities.addAll(cities2); // cities2=[New York, San Francisco, Moab, Atlanta, Memphis]

Similarly, the *ArrayList* has a constructor that accepts any type of *Collection* (of the same type or subtype) and initializes the list with those elements. As a review:

List<String> cities3 = **new** ArrayList<>(cities2); // cities3=[Atlanta, Memphis]

In this section, we consider the *retainAll, removeAll,* and *containsAll* methods defined in the *Collection* interface. In the next section we consider the *iterator* method. The *retainAll* method accepts any type of *Collection* (of the same type or subtype) and retains in its collection those elements that are in common with elements in the argument. Thus, it does a set intersection storing the result in the list that has had the method called on it. For example:

List<String> cities = **new** ArrayList<>(Arrays.*asList*("New York", "San Francisco", "Moab"));

List<String> cities2 = **new** ArrayList<>(Arrays.*asList*("Atlanta","Moab","New York", "Memphis"));

cities.retainAll(cities2); // cities=[New York, Moab]

If you didn’t want to modify the two existing lists: *cities* and *cities2,* then you could simply make a copy of one of the lists before using *retainAll.* For example:

List<String> cities = **...**

List<String> cities2 = **...**

List<String> citiesIntersection = **new** ArrayList<>(cities);

citiesIntersection.retainAll(cities2);

The *Collection* interface defines a *removeAll* method that accepts any type of *Collection* (of the same type or subtype) and removes from its collection those elements that are also in the argument. In other words, it removes from the first list the intersection with the second list. For example:

List<String> cities = **new** ArrayList<String>(Arrays.*asList*("New York", "San Francisco", "Moab"));

List<String> cities2 =**new** ArrayList< >(Arrays.*asList*("Atlanta","Moab","New York","Memphis"));

cities.removeAll(cities2); // cities=[San Francisco]

The *Collection* interface defines a *containsAll* method that accepts any type of *Collection* (of the same type or subtype) and returns *true* if the collection contains all the elements in the argument; and *false* otherwise. For example:

List<String> cities1 = **new** ArrayList<>(Arrays.*asList*("New York","San Francisco", "Moab"));

List<String> cities2 = **new** ArrayList<>(Arrays.*asList*("Atlanta","Moab","New York", "Memphis"));

List<String> cities3 = **new** ArrayList<String>(Arrays.*asList*("Moab", "New York"));

**boolean** isC2inC1 = cities1.containsAll(cities2); // isC2inC1=false

**boolean** isC3inC1 = cities1.containsAll(cities3); // isC3inC1=true

For the *retainAll, removeAll,* and *containsAll* methods to work correctly with a list of custom objects, *equals* must be overridden in the custom class.

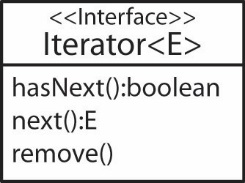
## Exercises

For the next several problems, suppose you have two *Lists* of jersey numbers (int) for two different basketball teams. The solutions for all these problems are in the *exercise\_jerseys* package.

1. Write a method, *commonJerseys* that accepts two lists of jersey numbers and returns a new *List* with all the jersey numbers that are in common. Hint: use *retainAll.* For example: if *jerseys1*={2,3,4,5}and *jerseys2*={1,5,3}*,* then the result of this method is a list with: {3,5}.
2. Write a method, *firstButNotSecond* that accepts two lists of jersey numbers and returns a new *List* with all the jersey numbers that are in the first list but not in the second. For example: if *jerseys1*={2,3,4,5} and *jerseys2*={1,5,3}*,* then the result of this method is a list with: {2,4}. (a) Solve using a loop. Hint: use *contains*. (b) Solve without using any loops. Hint: use *removeAll.*
3. Write a method, *mergeJerseys* that accepts two *Lists* of jersey numbers and returns a new list that combines the two lists but with no duplicates. You can assume that each list, individually, does not contain any duplicates; however, there could be duplicates between the lists. The method should not change either of the two input lists. For example: if *jerseys1*={2,3,4,5} and *jerseys2*={1,5,3}*,* then the result of this method is a list with: {2,3,4,5,1}. Hint: the easiest way is to use a loop and *contains*.
4. Write a method, *notInBoth* that accepts two lists of jersey numbers and returns a new *List* with all the jersey numbers that are in either the first list or the second list, but not both. For example: if *jerseys1*={2,3,4,5} and *jerseys2*={1,5,3}*,* then the result of this method is a list with: {2,4,1}. (a) Solve using two sequential (*i.e.* not nested) loops. Hint: use *contains*. (b) Solve without using any loops. Hint: usethe *firstButNotSecond* method from a previous problem twice and then put the two lists together to return.

# Iterators

The code for the examples in this section are in the *example\_iterator\_players* package.

The *Collection* interface defines an *iterator* method that returns an object that implements the *Iterator* interface, which is shown in the class diagram on the right. An *Iterator* object serves two functions: (a) a way to iterate over all the objects in a collection without having to know *how* they are stored in the collection. (b) a way to remove items from a collection while you are iterating over it.

## Iterating over a Collection with an Iterator

For the examples that follow, we will use a list of *Player* objects, where a *Player* has a *name* and a *score*.

ArrayList<Player> players = **new** ArrayList<>();

players.add(**new** Player("Pam", 24));

players.add(**new** Player("Len", 19));

players.add(**new** Player("Malia", 37));

players.add(**new** Player("Bob", 13));

players.add(**new** Player("Rea", 46));

To use an iterator to traverse this list:

Iterator<Player> iter = players.iterator();

**while**(iter.hasNext()) {

Player player = iter.next();

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

}

Note that the *Iterator* interface is generic, so we must specify what type of objects we are iterating over. The result of the code above is of course, no different from either of the approaches below:

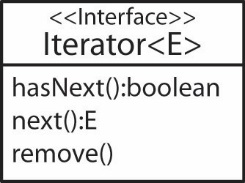
|  |  |  |
| --- | --- | --- |
| **for-each Loop** |  | **Indexed Loop** |
| **for**(Player player : players) {  System.***out***.println(player);  } |  | **for**(**int** i=0; i<players.size(); i++) {  System.***out***.println(players.get(i));  } |

Most people would not use an iterator to iterator over a collection, they would use the *for-each* or indexed loop. A good use for an iterator is to remove certain elements as we iterate over it, which we consider in the next section. Interestingly, the *for-each* loop compiles down to an iterator.

he *Iterator* interface is an excellent example of object-oriented design, which uses information hiding. An iterator hides the actual storage mechanism of the data. Thus, the user of an iterator does not need to know how the data is stored, it just knows that it can access the *next* item. For example, with the indexed loop, if you are using an *ArrayList*, you must use the *get* method, with other types of collections, there may be some other method to access an item, or no method at all. With an iterator, you don’t need to know these.

## Filtering a Collection with an Iterator – Removing Elements

*Filtering* a collection refers to the idea of finding all the elements in a collection that meet certain criteria. The way we consider it here, is we will remove all elements that meet the criteria from the collection. An *Iterator* is the preferred way to filter a collection by removing elements.

In the *players* list considered above, suppose we want to remove all players whose score is less than 20. In this case, we can use the *Iterator*’s *remove* method:

Iterator<Player> iter = players.iterator();

**while**(iter.hasNext() ) {

Player player = iter.next();

**if**(player.getScore() < 20) {

iter.remove();

}

}

A common mistake is to use a *for-each* loop and then using the collection’s *remove* method. However, such code will fail if the *remove* method is executed, throwing a *ConcurrentModificationException.* You cannot modify a collection (add to or remove from) while iterating over it with a *for-each* loop. An example is found in an [Appendix](#Appendix_filter_foreach_incorrect). Another common mistake is to use an indexed loop, somewhat naively. An example is found in an [Appendix](#Appendix_filter_indexed_incorrect). Other approaches to filtering that work are: an indexed loop with a subtle modification of the index inside the loop (bad practice), a *while* loop, an indexed loop traversing the list in reverse order, or using the *Stream*’s *filter* method. Examples of these are found in an [Appendix](#Appendix_filter_other_approaches).

## Filtering a Collection with an Iterator – Removing & Returning Elements

Suppose you want to remove certain elements from a collection and also return the removed elements in a new collection. Considering the example from above, suppose we want to (a) remove all players whose score is less than 20 from the *players* list and (b) put those removed players in another list named *lowScorePlayers.* Note that every time the *next* method is called, the next element in the collection is retrieved. Thus, if you call *next* twice inside the loop, you will receive the next two elements, respectively. Thus, if you need the next element more than once in the loop, you must store it in a variable. The correct version is shown in the table below on the left. There, we make one call to *next* inside the loop, capturing the item in the *player* variable. Then, *player* is used twice. Both examples define this list to store the players that are to be stored in a separate list:

ArrayList<Player> lowScorePlayers = **new** ArrayList<>();

|  |  |
| --- | --- |
| **Correct** | **Incorrect** |
| Iterator<Player> iter = players.iterator();  **while**( iter.hasNext() ) {  Player player = iter.next();  **if**(player.getScore() < 20) {  iter.remove();  lowScorePlayers.add(player);  }  }  // Result  players: [Pam-24, Malia-37, Rea-46]  lowScorePlayers: [Len-19, Bob-13] | Iterator<Player> iter = players.iterator();  **while**(iter.hasNext()) {  **if**(iter.next().getScore() < 20) {  iter.remove();  lowScorePlayers.add(iter.next());  }  }  // Result  players: [Pam-24, Malia-37, Rea-46]  lowScorePlayers: [Malia-37, Rea-46] |

For the incorrect version, consider the original list of players:

[Pam-24, Len-19, Malia-37, Bob-13, Rea-46]

When the player, “Len-19” is found, it is removed, but then the subsequent call to *iter.next()* advances to the next player, “Malia-13”, which is added to *lowScorePlayers.* Then, the loop repeats, where the first *next* retrieves: “Bob-13, which is removed and the subsequent call to *iter.next()* adds “Rea-46” to *lowScorePlayers*. So, we can see that (a) we are putting the wrong players in the list, (b) some players do not have their score checked – we are affectively skipping them, (c) if there has been a player with score less than 20 in the last space, then there is the possibility that the code would throw an exception as the coded tried to add the next element to *lowScorePlayers,* (d) the loop, in this case only executed 3 times. This is a common mistake. Thus, if you need the current element in the loop more than once, you should store it in a variable with a single call to *iter.next* as shown in the correct version above.

## Exercises

1. (Solution in *exercise\_cull\_dogs* package) Consider the *Dog* class below. Write a static method, *cullDogs* that accepts a list of *Dogs,* and an integer, *maxAge*, and removes dogs that are older than *maxAge* from the input list, and returns those removed dogs in new list.

**public** **class** Dog {

**private** **int** age;

**public** Dog(**int** age) {

**this**.age=age;

}

**public** **int** getAge() {

**return** age;

}

@Override

**public** String toString() {

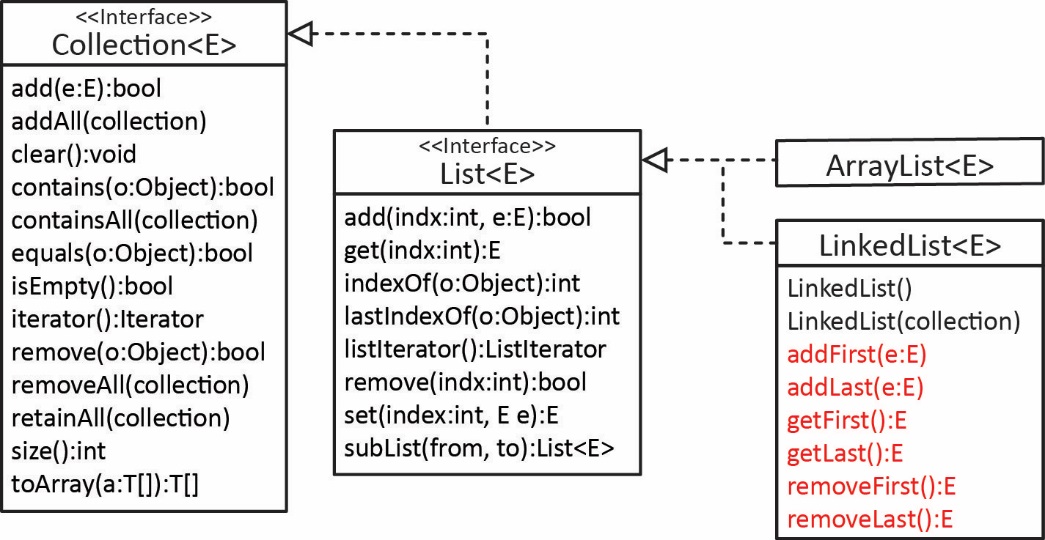
**return** "age=" + age;

}

}

# The *LinkedList* Class

The *LinkedList* class, shown in the class diagram below is very similar to the *ArrayList* class in that they both implement the *List* and *Collection* interfaces. Almost anything you can do with an *ArrayList,* you can do with a *LinkedList*. Internally, they are implemented differently. This is important because *LinkedList* can be faster in some situations, which we will see in an example later.



In addition, the *LinkedList* class introduces a few *convenience methods[[1]](#footnote-1)* (shown in red above) for operating on the first and last elements in the collection. They are called *convenience methods* because their result can be achieved by existing *List* interface methods, except that they are easier to use. For example:

|  |  |
| --- | --- |
| ***LinkedList*** | ***List*** |
| list.addFirst(x) | list.add(0,x) |
| list.addLast(x) | list.add(x) |
| list.getFirst() | list.get(0) |
| list.getLast() | list.get(list.size-1) |
| list.removeFirst() | list.remove(0) |
| list.removeLast() | list.remove(list.size-1) |

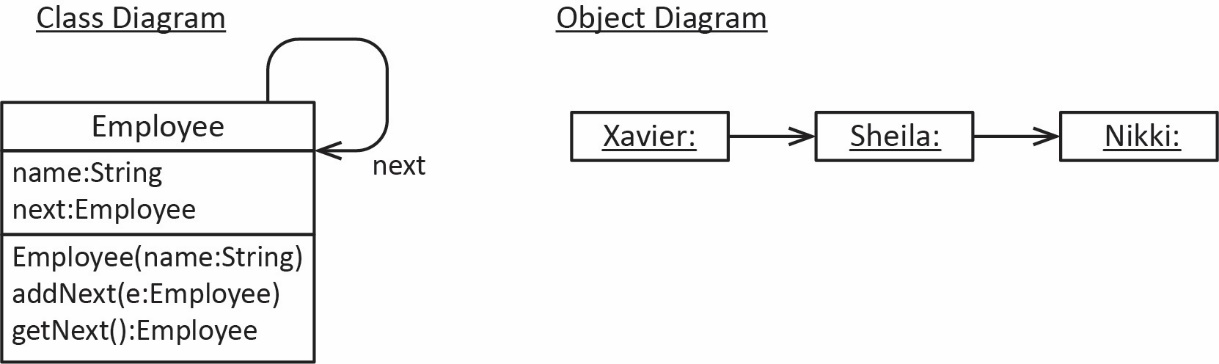
Consider the example from Lab 11 where did an experiment to see how long it took to add elements into the first position to an *ArrayList* and a *LinkedList.* We started with an *ArrayList* that initially contained 100,000 random integers. Next, we timed how long it took to insert 100,000 more random integers into the first position into this *ArrayList*. Then, we repeated for *LinkedList.* The results on my computer were:

ArrayList size: 100000, time to add 100000 vals: 3.644 sec

LinkedList size: 100000, time to add 100000 vals: 0.006 sec

Thus, in this one example, the *ArrayList* took more than 600 times as long as *LinkedList*.

In general, a linked list is a class with an association with itself (in the simplest case). For example:



For example (code in *example\_custom\_linkedlist* package), the *Employee* class is shown below where we see that it has an instance variable of type *Employee.* In other words, an *Employee has-a Employee*; they form a linked list. You will learn about this in CS 3410 (Data Structures).

|  |  |
| --- | --- |
| **Class** | **Example** |
| **public** **class** Employee {  **private** String name;  **private** Employee next;    **public** Employee(String name) {  **this**.name = name;  }    **public** **void** addNext(Employee e) {  **this**.next = e;  }    **public** Employee getNext() {  **return** next;  }  ...  } | Employee e1 = **new** Employee("Xavier");  Employee e2 = **new** Employee("Sheila");  e1.addNext(e2);  Employee e3 = **new** Employee("Nikki");  e2.addNext(e3);  Employee e = e1;  **while**(e != **null**) {  System.***out***.println(e);  e = e.getNext();  } |

As we stated earlier, an *ArrayList* is backed by an array. A *LinkedList* uses an approach similar to above where space is allocated in memory (the *heap*) dynamically.

How to choose between *ArrayList, LinkedList,* and *Array –* performance characteristics

* Prefer *ArrayList* when you need positional (random) access without adding or removing from the beginning of the list. In CS 3410 you will learn that: *add, remove, contains* are O(n) and *get* is O(1).
* Prefer *LinkedList* when you need to add or remove from the beginning of a list. In CS 3410 you will learn that: *add* and *remove* are O(1) and *contains* and *get* are O(n).
* Prefer *Array* when you don’t need to add or remove elements at all. In other words, you have a fixed list.

# The *Comparator* Interface

In another chapter, we saw how we could sort a list containing instances of a custom class by having the class implement *Comparable*, by providing a *compareTo* method. This allowed us to sort on one set of criteria. However, what if we want to sort on different criteria at different times? For example, suppose we want to sort a list of *Employee* objects based on name, and then later based on SSN, and then later on salary. A solution is to use the *Comparator* interface.

With this approach, we define a separate *Comparator* class for each different ordering. Then, the *Collections* class has an overloaded *sort* method that accepts a list and a comparator. For example, we can define a list of *Employees:*

List<Employee> employees = **new** ArrayList<>();

Employee e1 = **new** ...

...

employees.add(e1);

Then create a name comparator (considered shortly) and supply it to the *sort* method:

EmployeeNameComparator compName = **new** EmployeeNameComparator();

Collections.sort(employees, compName);

And later create an SSN comparator and supply it to the *sort* method:

EmployeeSSNComparator compSSN = **new** EmployeeSSNComparator();

Collections.sort(employees, compSSN);

And finally create a salary comparator and supply it to the *sort* method:

EmployeeSalaryComparator compSalary = **new** EmployeeSalaryComparator();

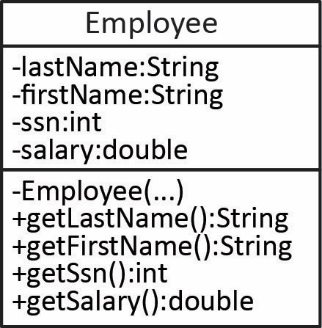
Collections.sort(employees, compSalary);

Thus, *Comparator* provides a more flexible way of ordering objects as multiple comparators can be defined to order objects in different ways.

*Comparator* is a generic interface in Java (shown below on left) that is used to compare two objects to see which one is “larger”, “smaller”, or if they are “equal”. In this sense, it is very similar to the *Comparable* interface. The *compare* method accepts two arguments of generic type *T* and the return is defined in the table below on the right:

|  |  |
| --- | --- |
| ***Comparator* Interface** | **Return values for *compare* method** |
| G:\eDataClasses\CS 1302 - Programming 2\notes\10_Ch 20 - List, Stack, Q, Priorty Q\a3.jpg | |  |  | | --- | --- | | **Return** | **Condition** | | Negative Integer | o1<o2 | | 0 | o1=o2 | | Positive Integer | o1>o2 | |

Thus, we simply create a class that implements *Comparator* and provide a *compare* method that defines how to compare two objects.

For example (code in *example\_comparator\_employee* package), consider the *Employee* class shown on the right. We can define a *Comparator* to order on SSN’s.

**public** **class** EmployeeSSNComparator **implements** Comparator<Employee> {

**public** **int** compare(Employee e1, Employee e2) {

**return** e1.getSSNum() - e2.getSSNum();

}

}

We can define another *Comparator* to order on names (last, then first).

**public** **class** EmployeeNameComparator **implements** Comparator<Employee> {

**public** **int** compare( Employee e1, Employee e2 ) {

**int** diff = e1.getLastName().compareTo(e2.getLastName());

**if**( diff != 0 ) {

**return** diff;

}

**else** {

**return** e1.getFirstName().compareTo(e2.getFirstName());

}

}

}

Finally, we can define a *Comparator* to order *Employees* on their salaries:

**public** **class** EmployeeSalaryComparator **implements** Comparator<Employee> {

**public** **int** compare( Employee e1, Employee e2 ) {

**double** diff = e1.getSalary() - e2.getSalary();

**if**(diff < 0.0) {

**return** -1;

}

**else** **if**(diff > 0.0) {

**return** 1;

}

**else** {

**return** 0;

}

}

}

Finally, we can use the comparators:

// Build a list of employees

List<Employee> employees = *buildEmployeesList*();

// Create a new list with original employees

List<Employee> empsName = **new** ArrayList<>(employees);

EmployeeNameComparator compName = **new** EmployeeNameComparator();

// Sort new list based on name

Collections.*sort*(empsName, compName);

List<Employee> empsSSN = **new** ArrayList<>(employees);

EmployeeSSNComparator compSSN = **new** EmployeeSSNComparator();

Collections.*sort*(empsSSN, compSSN);

List<Employee> empsSalary = **new** ArrayList<>(employees);

EmployeeSalaryComparator compSalary = **new** EmployeeSalaryComparator();

Collections.*sort*(empsSalary, compSalary);

The *binarySearch* static method of the *Collections* class is used to search for an item in a collection. The list must be sorted for this method to work. The general syntax is:

**int** pos = Collections.*binarySearch*(list, objectToSearchFor, comparator);

Note the following about the parameters:

|  |  |
| --- | --- |
| **Variable** | **Description** |
| *list* | List to be searched. Must have previously been sorted using a *Comparator*. |
| *objectToSearchFor* | The object we are searching for. But why are we searching for an object if we already have it? The idea is that we may only know the SSN for Employee and want to find the complete *Employee* object with a matching SSN. To do this, we will create a *dummy* employee with the information we have. For example, suppose we know the SSN (243558673) of the employee we want to look for, we would create a dummy *Employee* like this:  Employee eKey = **new** Employee("don't know", 243558673, -9999.0);  Notice that we simply made up a value for the name and salary. |
| *comparator* | This specifies how we want to compare objects. For example, the binary search algorithm needs to know if we are looking for an *Employee* based on SSN, or name, or salary. |
| *pos* | Returns the location of the item if it is found or a negative integer if not found, which is exactly the same as the *indexOf* method in the *ArrayList* class except that with *indexOf,* it returns -1 if the item is not found. *binarySearch* returns a negative integer, which has meaning, but we won’t consider it here. |

Let’s summarize how to use binary search for the situation where we are looking for an *Employee* object in a *List* with a matching SSN (code in *example\_comparator\_employee* package):

1. Ask the user what the SSN is for the *Employee* object they want to search for. (Suppose that value is: 243558673).
2. Next, we create a *dummy* employee using just the information we have (the SSN):

Employee eKey = **new** Employee("don't know", 243558673, -9999.0);

1. Make sure list is sorted according to SSN:

EmployeeSSNComparator ssnComp = **new** EmployeeSSNComparator();

Collections.*sort*(employees, ssnComp);

1. Do the binary search:

**int** pos = Collections.*binarySearch*(employees, eKey, ssnComp);

1. Check to see if *Employee* was found. If so, *get* them employee and print.

**if**(pos >= 0) {

Employee e = employees.get(pos);

System.***out***.println( "Employee found : " + e);

}

**else**{

System.***out***.println("\*\*\* Employee Not Found \*\*\*, pos=" + pos);

}

## Exercises

1. (Solution in *exercise\_blobs* package) Consider the *Blob* class below.
2. Write a *BlobAgeComparator* that orders *Blobs* based on *age*, ascending. Write a test method(s) in a *BlobComparatorTest* class (we will use the same test class for all the methods that test comparators).
3. Write a *BlobVolumeComparator* that orders *Blobs* based on *volume*, ascending. Write a test method(s) in a *BlobComparatorTest* class
4. Write a *BlobAgeVolumeComparator* that orders *Blobs* based on *age,* ascending, then *volume*, ascending. Write a test method(s) in a *BlobComparatorTest* class

**public** **class** Blob {

**private** **int** age;

**private** **double** volume;

**public** Blob(**int** age, **double** volume) {

**this**.age = age;

**this**.volume = volume;

}

**public** **int** getAge() {

**return** age;

}

**public** **double** getVolume() {

**return** volume;

}

@Override

**public** String toString() {

**return** "(age=" + age + ":vol=" + volume +")";

}

}

1. (Solution in *exercise\_blobs* package) Consider the *Blob* class from the previous problem, and the *BlobManager* class below.
2. Write a method, *getYoungest* that returns the *Blob* with the least age. Hint: you don’t need a loop, use a method from the *Collections* class. snippet of code to obtain a reference to the *Blob* with the least *age*.
3. Write a method, *getBlobsOnAge* that returns a list of *Blob* object that are sorted on *age.* This method should not alter the original the order in the *blobs* instance variable.
4. Write a method, *getSortedBlobs* that accepts a string, *sortType*. The valid values of *sortType* and the appropriate return is shown in the table below. This method should not alter the original the order in the *blobs* instance variable.

|  |  |
| --- | --- |
| ***sortType*** | **Description** |
| “age” | The method returns a list of *Blob* object that are sorted on *age* |
| “vol” | The method returns a list of *Blob* object that are sorted on *volume* |
| “agevol” | The method returns a list of *Blob* object that are sorted on *age*, followed by *volume* |

1. Write a snippet of code to sort the *Blobs* on *age* ascending, followed by *volume* ascending

**public** **class** BlobManager {

**private** List<Blob> blobs = **new** ArrayList<>();

**public** BlobManager() {}

**public** **int** getNumBlobs() {

**return** blobs.size();

}

**public** **void** addBlob(Blob b) {

blobs.add(b);

}

**public** Blob getBlob(**int** i) {

**if**((i >= 0) && (i < blobs.size())) {

**return** blobs.get(i);

}

**return** **null**;

}

**public** String toString() {

String msg = "";

**for**(**int** i=0; i<getNumBlobs(); i++) {

String acnt = String.*format*("%d - %s\n", i+1, blobs.get(i));

msg += acnt;

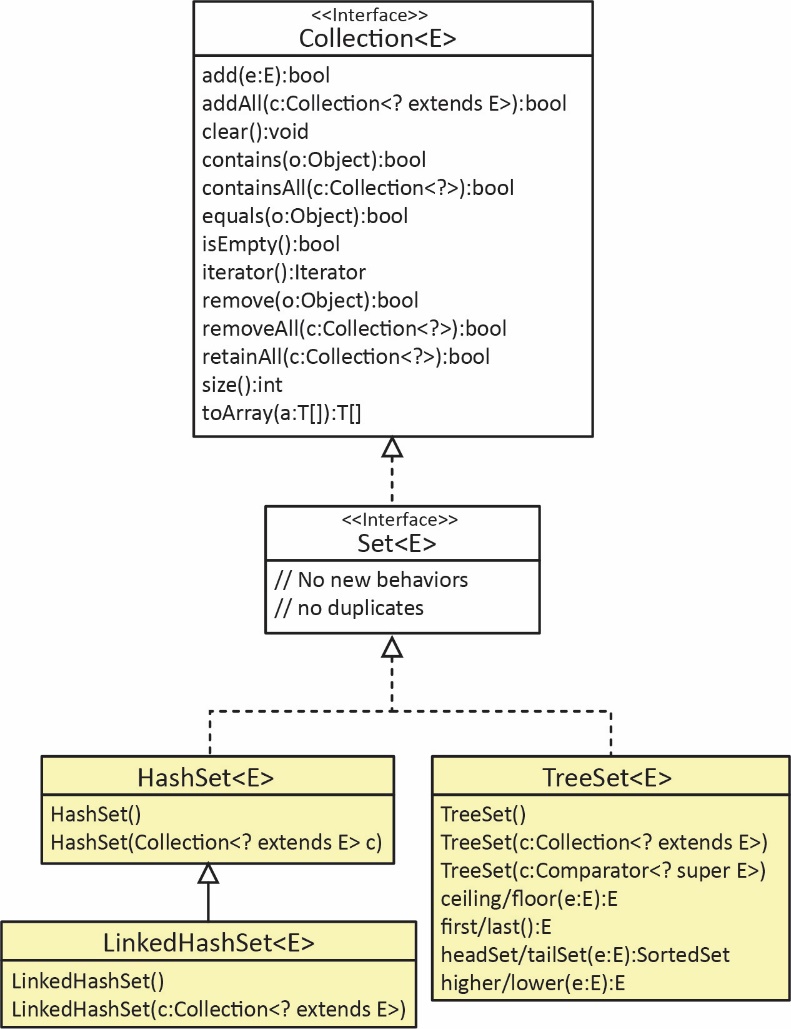
}

**return** msg;

}

}

# The *Set* Interface

*Set* is a sub-interface of *Collection* that:

1. Doesn’t allow duplicates.

* There are no two elements: e1 and e2 in the set such that e1.equals(e2).
* mySet.add(e1) simply returns false if e1 already exists.

1. Does not specify any additional methods. *Set* is a (somewhat) [marker interface](http://jtechies.blogspot.com/2012/07/item-37-use-marker-interfaces-to-define.html)*.*
2. Doesn’t provide random (positional) access.

* The *Collection* interface does not specify a *get(pos)* method.
* The only way to access the elements is by using a *for-each* loop, or an *iterator*.

Java provides three common implementations:

1. *HashSet* – Doesn’t guarantee any particular ordering. If you iterate over a *HashSet*, you will see all the elements, but they will not be (in general) in the order that you added them.
2. *LinkedHashSet* – Elements are ordered according to the order they were added.
3. *TreeSet* – Elements are ordered according to *Comparable* or *Comparator*.

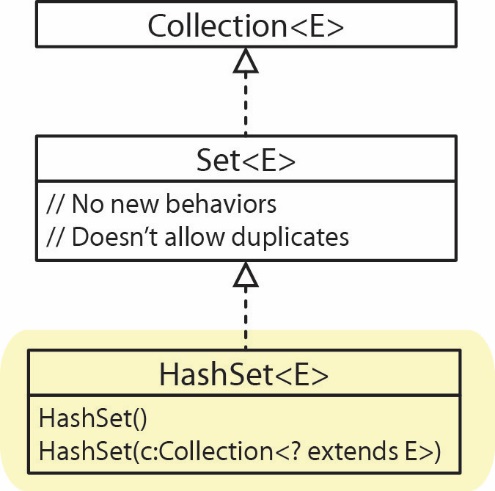
The *Set* classes are extremely fast, in fact, there is nothing faster for some operations. Consider the *add*, *remove*, *contains* methods.

1. *HashSet* – Very fast, O(1)\*. Essentially a constant amount of time no matter how many items are in the set.
2. *LinkedHashSet* – Very fast, O(1)\*
3. *TreeSet* – Fast, O(log n)\*

\* This is called *Big ‘O’* notation. You will learn about this in a data structures course (CS 3410). It is a measure of howfast an algorithm is. We will briefly discuss the graph at the top of this page: <http://bigocheatsheet.com/>. Another reference:

<http://infotechgems.blogspot.com/2011/11/java-collections-performance-time.html>

# The *HashSet* Class

The only methods that *HashSet* provides are *Collection* methods. It has two constructors:

1. A no-arg constructor, for example:

Set<String> hsCities = **new** HashSet<>();

1. One that accepts any type of *Collection* (of the same type or subtype) and initializes the set with those elements*.* For example:

ArrayList<String> alCities = **new** ArrayList<>();

...

Set<String> hsCities = **new** HashSet<>(alCities);

We can iterate over a *HashSet* using a *for-each* loop or an *iterator.* As we noted earlier, there is no guarantee of order. For example (code in *example\_hash\_set* package):

Set<String> names = **new** HashSet<>();

names.add("cat"); names.add("dab"); names.add("fia");

names.add("fre"); names.add("gor"); names.add("pet");

|  |  |
| --- | --- |
| **For-each loop** | **Iterator** |
| **for**(String name : names) {  System.***out***.print(name + " ");  }  System.***out***.println(); | Iterator<String> iter = names.iterator();  **while**(iter.hasNext()) {  System.***out***.print(iter.next() + " ");  } |
| Output: dab cat fre gor pet fia | Output: dab cat fre gor pet fia |

As stated earlier, we cannot use an indexed loop, as a *Set* is not a *List*, *i.e.* there is no positional access with a *Set*.

As mentioned previously, f*iltering* a collection refers to the process of iterating over the collection and selectively removing (collecting, *etc.*) certain elements and the preferred way to do that is to use an *Iterator*. For example, if we have a set of names, a snippet of code to remove names that contain the letter, “a”:

Set<String> names = **new** HashSet<>();

...

Iterator<String> iter = names.iterator();

**while**(iter.hasNext()) {

**if**(iter.next().contains("a"))

iter.remove();

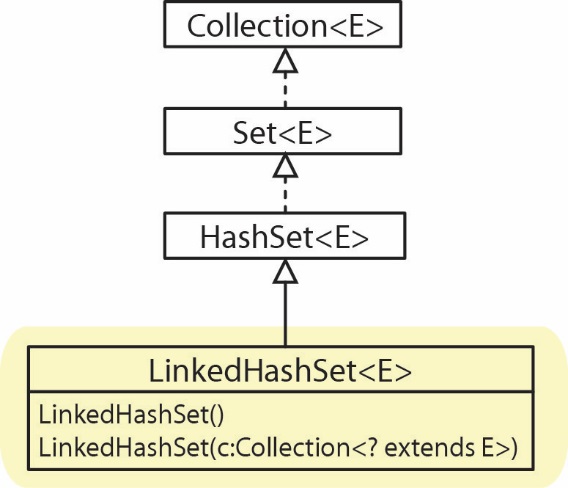
}

We can store instances of a custom class in a *HashSet*, but you must override *hashCode* and *equals*. We will not consider this in this class; however, you will learn what a *hash code* is in CS 3410[[2]](#footnote-2) [[3]](#footnote-3).

## Exercises

1. (Solution in *exercise\_names* package) Write a method, *removeLongNames* that accepts a set of names and an integer, *len.* The method should remove any names from the set with length greater than *len.* See [Section 3.3](#_Filtering_a_Collection) if needed.
2. (Solution in *exercise\_names* package) Write a method, *separateLongNames* that accepts a set of names and an integer, *len.* The method should remove any names from the set with length greater than *len* and return a set of the names that were removed.See [Section 3.3](#_Filtering_a_Collection) if needed.
3. (Solution in *exercise\_names* package) Write a method, *getUniqueNames* that accepts a list of names and returns a list with duplicates removed, *e.g.* Example: *getUniqueNames(“alpha”, “beta”, “alpha”, “gamma”} -> {“alpha”, “beta”,”gamma”}*. Hint: use a *HashSet* and before returning, convert it to a list.

# The *LinkedHashSet* Class

*LinkedHashSet* is a subclass of *HashSet* as shown in the class diagram on the right. *LinkedHashSet* is identical to *HashSet* except that the order of insertion is preserved. For example (code in *example\_linked\_hash\_set* package):

**private** **static** **void** testLinkedHashSet() {

Set<String> names = **new** LinkedHashSet<>();

names.add("cat"); names.add("dab");

names.add("fia"); names.add("fre");

names.add("gor"); names.add("pet");

**for**(String name : names) {

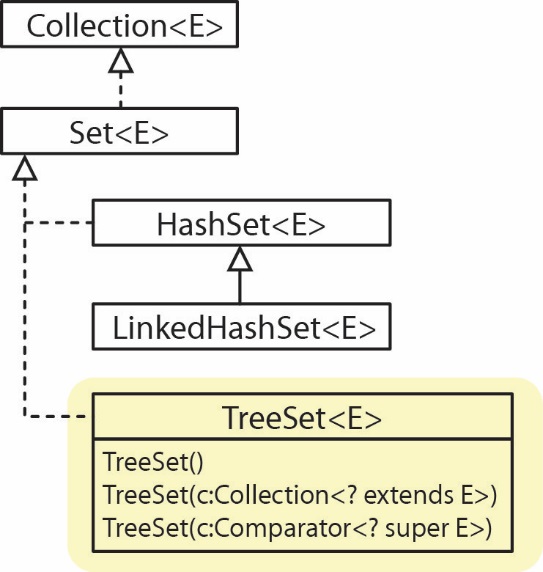
System.***out***.print(name + " ");

}

}

Output: cat dab fia fre gor pet

# The *TreeSet* Class

The *TreeSet* class is an implementation of the *Set* interface as shown in the class diagram on the right. A *TreeSet* is an ordered set where elements are ordered according to *Comparable* or *Comparator*. A *TreeSet* can be created with no arguments, a *Collection*, or a *Comparator*.

The *TreeSet* class does inherit other methods, which are considered in an [appendix](#Appendix_sorted_set).

Example (code in *example\_tree\_set* package) – A *TreeSet* of *Strings*

TreeSet<String> tsCities = **new** TreeSet<>(

Arrays.*asList*("New York", "Atlanta",

"Savannah", "Tampa", "Durango"));

**for**(String city : tsCities) {

System.***out***.print(city + " ");

}

Output: Atlanta Durango New York Savannah Tampa

## Exercises

1. (Solution in *exercise\_names* package) Write a method, *getUniqueNames2* that accepts a list of names and returns a set with duplicates removed and ordered alphabetically, *e.g.* Example: *getUniqueNames2(“Leno”, “Rich”, “Leno”, “Kate”} -> {“Billie”, “Kate”, ”Rich”}*.
2. (Solution in *exercise\_get\_words\_alphabetic* package) Write a method, *getWordsAlphabetic* which accepts a comma-delimited string of words, *words* and returns a set of the words, with no duplicates, ordered alphabetically. For example:

*getWordsAlphabetic("beetle,dog,beetle,ant,cat,ant")*

returns the set: [ant, beetle, cat, dog].

1. (Solution in *exercise\_get\_domains\_alphabetic* package) Write a method, *getDomainsAlphabetic* which accepts a comma-delimited string of email addresses and returns a list of domains alphabetically with no duplicates. The format for an email address is: local-part@domain. For example, if the input is:

"earl@inorbit.com,bip@run.com,arf@run.edu,cam@inorbit.com,dg@gmail.com,repo@ant.edu”

then the returned list is:

[ant.edu, gmail.com, inorbit.com, run.com, run.edu]

Hint: very similar to the last problem except that you have to parse off the domain, and return a list instead of a set.

1. (Solution in *exercise\_get\_paid\_list* package) You will write two methods, both accept a list of names and a set of integers. The integers represent indices in the list of names of names that have paid their bill.

|  |  |
| --- | --- |
| **Method** | **Description** |
| *getPaidList(names, indices)* | Returns a list of names that have paid their bill. |
| *getNotPaidList(names, indices)* | Returns a list of names that have not paid their bill. |

You must not change the input list nor set. For example, suppose the input is: names=[t,b,d,g,m,e] and iPaid=[4,1,2]; then the *getPaidList* returns: [b,d,m] and *getNotPaidList* returns: [t,g,e].

Consider this method: *getPaidList(names, indices)* – Returns a list of names that have paid their bill while also removing those from the list. This is a little harder problem because when you remove from the list of name, the subsequent names in the list have their indices renumbered. No solution is provided for this method.

# The *TreeSet* Class with Custom Objects

A *TreeSet* can hold instances of a custom class provided the class implements *Comparable* or a *Comparator* exists which is supplied in the constructor. For example (code in *example\_treeset\_players* package):

1. *TreeSet* with *Comparable*

|  |  |
| --- | --- |
| **Class** | **Example** |
| **public** **class** Player **implements** Comparable<Player> {  **private** String name;  **private** **int** score;  ...  @Override  **public** **int** compareTo(Player p) {  **return** **this**.name.compareTo(p.name);  }  } | TreeSet<Player> players = **new** TreeSet<>();  players.add(**new** Player("zeke", 300));  players.add(**new** Player("ben", 100));  players.add(**new** Player("alan", 400));  **for**(Player p : players)  System.***out***.println(p);  **Output**  Name=alan, Score=400  Name=ben, Score=100  Name=zeke, Score=300 |

1. *TreeSet* with *Comparator*

|  |  |
| --- | --- |
| **Comparator (same class as above)** | **Example** |
| **public** **class** PlayerScoreComparator  **implements** Comparator<Player> {  @Override  **public** **int** compare(Player p1, Player p2) {  **return** p1.getScore()-p2.getScore();  }  } | TreeSet<Player> players =  **new** TreeSet<>(**new** PlayerScoreComparator());  players.add(**new** Player("zeke", 300));  players.add(**new** Player("ben", 100));  players.add(**new** Player("alan", 400));  **for**(Player p : players)  System.***out***.println(p);  **Output**  Name=ben, Score=100  Name=zeke, Score=300  Name=alan, Score=400 |

Next, we consider (code in *example\_treeset\_employees* package) using the *contains* and *remove* methods both of which accept an *Object* as an argument:

1. Suppose we have an *Employee* class:

**public** **class** Employee {

**private** String lName;

**private** String fName;

**private** **int** ssn;

**private** **double** salary;

**public** Employee(String lName, String fName, **int** ssNum, **double** salary) {

...

}

...

}

1. And a comparator that orders on SSN:

**public** **class** EmployeeSSNComparator **implements** Comparator<Employee> {

**public** **int** compare(Employee e1, Employee e2) {

**return** e1.getSSNum() - e2.getSSNum();

}

}

1. Then, we can create a *TreeSet* of *Employee*s using the comparator:

TreeSet<Employee> emps = **new** TreeSet<>(**new** EmployeeSSNComparator());

Employee e1 = **new** Employee("Boggs", "Kay", 716533892, 12.57);

Employee e2 = **new** Employee("Lyton", "Ben", 476227851, 77.88);

Employee e3 = **new** Employee("Boggs", "Amy", 553572246, 22.32);

Employee e4 = **new** Employee("Dern", "Donald", 243558673, 23.44);

emps.add(e1); emps.add(e2); emps.add(e3); emps.add(e4);

1. Write a static method, *contains* that accepts (a) a *TreeSet* of *Employee* objects (as described above, *i.e.* with a *Comparator* that orders on SSN) and (b) an SSN. This method returns *true* if there is an *Employee* in the set with SSN equal to the input SSN and *false* otherwise.

**private** **static** **boolean** contains(TreeSet<Employee> emps, **int** ssn) {

Employee key = **new** Employee("", "", ssn, 0);

**return** emps.contains(key);

}

Similarly, we could write a static method, *remove* that accepts (a) a *TreeSet* of *Employee* objects and (b) an SSN. This method removes the *Employee* in the set with SSN equal to the input SSN if it exists. If the remove is successful, it should return *true*, otherwise, *false*.

**private** **static** **boolean** remove(TreeSet<Employee> emps, **int** ssn) {

Employee key = **new** Employee("", "", ssn, 0);

**return** emps.remove(key);

}

## Exercises

1. (Solution in *exercise\_get\_ids* package) Consider the *Student* class below. Write a static method, *getIds* which accepts two *TreeSets* of *Student* objects and returns a list of the ids for the students that are in both sets (not a list of the students, a list of the *ids*).

**public** **class** Student **implements** Comparable<Student> {

**private** **int** id;

**private** **double** score;

Student(**int** id, **double** score) {

**this**.id = id;

**this**.score = score;

}

**int** getId() {

**return** id;

}

**double** getScore() {

**return** score;

}

**public** **int** compareTo(Student s) {

**return** **this**.getId() - s.getId();

}

**public** String toString() {

**return** "id=" + id + " score=" + score;

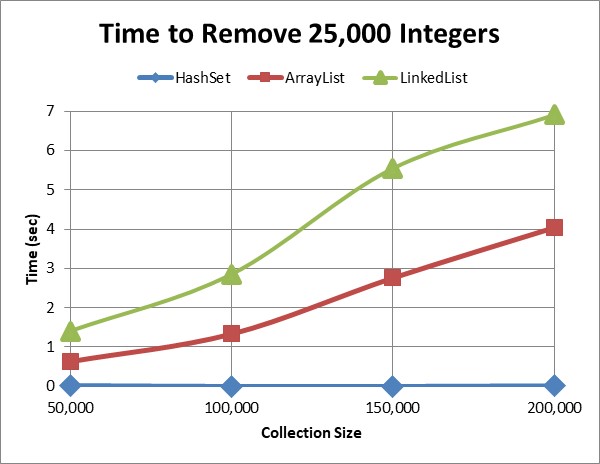
}

}

# Performance of Sets & Lists

I wrote some code (in the *example\_speed\_comparison* package) to compare how long it took to remove integers from *HashSet, LinkedHashSet, TreeSet, ArrayList,* and *LinkedList*. First 50,000 unique integers were generated and stored in a each of the collections. These are the collections where we will be removing from. Next, 25,000 of these values were selected at random. These are the values we will be removing from each of the collections. Finally, we timed how long it took to remove the 25,000 from the 50,000 in each of the collections. This process was repeated for collections of size: 100,000; 150,000; and 200,000 (continuing to remove 25,000 from each). The results are shown in the graph below.

The graph does not show *LinkedHashSet* nor *TreeSet* because the results were almost identical to *HashSet*. The interesting point is that we see that with *ArrayList* and *LinkedList,* the time required increases as the collection size increases whereas the there is no difference in time for the *HashSet,* which is also theoretical result.



The numerical results are shown below (seconds):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Collection Size** | **HashSet** | **LinkedHashSet** | **TreeSet** | **ArrayList** | **LinkedList** |
| 50,000 | 0.007 | 0.010 | 0.017 | 0.629 | 1.393 |
| 100,000 | 0.003 | 0.004 | 0.012 | 1.335 | 2.844 |
| 150,000 | 0.003 | 0.003 | 0.015 | 2.765 | 5.544 |
| 200,000 | 0.006 | 0.006 | 0.018 | 4.049 | 6.907 |

# Example: Counting Keywords

Consider a situation where you need to count and/or identify the occurrences of certain keywords in a body of text. For example, in a security scenario, suppose you want to scan a person’s Twitter feed for the occurrences of certain words that might indicate the person had a tendency towards violence.

In the example (code in *example\_counting\_happywords\_set*) that follows, we write methods to read the [text](https://cs.valdosta.edu/~dgibson/courses/cs1302/text/Catcher_in_the_Rye.txt) of the novel, [*The Catcher in the Rye*](https://en.wikipedia.org/wiki/The_Catcher_in_the_Rye)to: (a) count the number of “happy words” that occur in the text, (b) return a set of happy words in the order they occur in the text, (c) a sorted list of happy words that occur in the text.

Consider the following set of “happy words”:

**static** 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"};

**static** Set<String> *happyWords* = **new** HashSet<>(Arrays.*asList*(*happyWordsAry*));

1. An algorithm to count the number of occurrences of happy words in the text:

Create a HashSet with all happy words

Loop over all words in file

If word is in happy words set

Increment count

Return count

Next, we write a method that accepts a *File* object that references a text file and a *Set* of happy words. This method should return this count.

**public** **static** **int** countKeywords(File file, Set<String> happyWords) **throws** Exception {

**int** count = 0;

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

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

String word = input.next();

word = *getCleanWord*(word);

**if** (happyWords.contains(word))

count++;

}

input.close();

**return** count;

}

Where, *getCleanWord* simply removes any leading or trailing blanks spaces, converts the word to lower-case, and removes any punctuation or double quotes:

**public** **static** String getCleanWord(String word) {

String punctuation = ".,;:\"?!";

String cleanWord = word.trim();

cleanWord.toLowerCase();

**if**(cleanWord.length()>0) {

String lastChar = String.*valueOf*(cleanWord.charAt(cleanWord.length()-1));

**if**(punctuation.contains(lastChar)) {

cleanWord = cleanWord.substring(0,cleanWord.length()-1);

}

}

**return** cleanWord;

}

1. Next, we write a method that accepts a *File* object that references a text file and a *Set* of happy words. This method returns a set of happy word used in the text, in the order they appear. Thus, it is similar to the previous method.

**public** **static** Set<String> getWordsUsedInOrder(File file, Set<String> happyWords) **throws** FileNotFoundException {

Set<String> happyWordsUsed = **new** LinkedHashSet<>();

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

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

String word = input.next();

word = *getCleanWord*(word);

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

**if**(!happyWordsUsed.contains(word)) {// not nec., but makes more understandable

happyWordsUsed.add(word);

}

}

}

input.close();

**return** happyWordsUsed;

}

1. Finally, we write a method that is the same as the previous one, except that it returns a set of happy word used in the text, in alphabetical order. The solution is simple, we replace the *LinkedHashSet* that stores the result with a *TreeSet:*

Set<String> happyWordsUsed = **new** TreeSet<>();

Appendix

1. Filtering a Collection Incorrectly
   1. Filtering a Collection with a *for-each* Loop Incorrectly

This code below is very simple and obvious, and it will compile. However, ***if*** the *remove* method is executed (which it will in this example) then a *ConcurrentModificationException* will be thrown. We remember from a previous chapter that when using a *for-each* loop the corresponding collection cannot be modified (added to or removed from). Thus, this is not a viable approach for removing elements from a collection.

**for**(Player player : players) {

**if**(player.getScore() < 20) {

players.remove(player);

}

}

* 1. Filtering a Collection with a Forward, Indexed Loop Incorrectly

The code for the examples in this section are in the *example\_iterator\_cities* package.

Consider a list of strings and we want to remove all occurrences of “New York”:

ArrayList<String> cities = **new** ArrayList<>(Arrays.*asList*("Dallas", "New York", "New York", "San Fran", "Madison"));

Filtering a collection using an indexed loop, advancing forward through the list, will sometimes work incorrectly. For example:

**for**(**int** i=0; i<cities.size(); i++) {

**if**(cities.get(i).equals("New York")) {

cities.remove(i);

}

}

It will not work correctly if the list has duplicates and at least two are side-by-side. For example, consider the list of cities in the list in previous examples:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | … |
| “Dallas” | “New York” | “New York” | “San Francisco” | “Madison” |  |

With the loop above, at the conclusion, *cities* will still contain the second New York. The reason is that when the first New York is removed, all the cities to the right are moved over one position to the left and re-indexed. Thus, when *i=1*, the first New York is removed, but then the second New York is moved over and is now at index 1. Then, the next iteration begins at *i=2,* thus, skipping the second New York. For example:

i=1, beginning of loop:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | … |
| “Dallas” | “New York” | “New York” | “San Francisco” | “Madison” |  |

i=2, beginning of loop:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | … |
| “Dallas” | “New York” | “San Francisco” | “Madison” |  |

1. Other Approaches to Filtering a Collection

Consider a list of strings and we want to remove all occurrences of “New York”:

ArrayList<String> cities = **new** ArrayList<>(Arrays.*asList*("Dallas", "New York", "New York", "San Fran", "Madison"));

An approach to removing, that works is to use an indexed loop, forward over the elements, but decrement the index of the loop after any remove. This is done so that the loop doesn’t skip over an element when it removes. For example:

**for**(**int** i=0; i<cities.size(); i++) {

**if**(cities.get(i).equals("New York")) {

cities.remove(i--); // decrement i after removing, so doesn't skip

}

}

This is a poor approach because it is harder to understand and easier to get wrong. Best practice is to never change the index of a *for* loop inside the body of the loop itself. If an index needs to change in a loop, then it is preferred to use a *while* loop:

**int** i=0;

**while**(i < cities.size()) {

**if**(cities.get(i).equals("New York")) {

cities.remove(i);

}

**else** {

i++;

}

}

Another approach is to use an indexed loop to iterate over the elements in reverse order. For example:

**for**(**int** i=cities.size()-1; i>=0; i--) {

**if**(cities.get(i).equals("New York")) {

cities.remove(i);

}

}

This works because removing an element does not affect any of the elements before the element that was removed. Many people might choose this approach. I would consider this approach more subject to programmer error in the specification of the loop parameters as compared to the iterator approach.

To see that the indices before a removed element are not affected, consider this example: when i=4, and i=3, no change to the list is made. Now, see what happens when i=2:

i=2, beginning of loop (will remove “New York” at i=2):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | … |
| “Dallas” | “New York” | “New York” | “San Francisco” | “Madison” |  |

i=1, beginning of loop (will remove “New York” at i=1):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | … |
| “Dallas” | “New York” | “San Francisco” | “Madison” |  |

i=0, beginning of loop (no change):

|  |  |  |  |
| --- | --- | --- | --- |
| 0 | 1 | 2 | … |
| “Dallas” | “San Francisco” | “Madison” |  |

Another approach is to use the *Collection*’s *removeIf* method:

cities.removeIf(x -> x.equals("New York"));

Another approach is to use the *Stream*’s *filter* method to collect the items that meet a set of criteria and then use *removeAll:*

List<String> duplicates = cities

.stream()

.filter(x -> x.equals("New York"))

.collect(Collectors.*toList*());

cities.removeAll(duplicates);

1. Static Methods for Lists & Collections

The Collections class contains static methods for operating on Collections and Lists (and Maps). A few are listed below.

|  |  |
| --- | --- |
| **Method** | **Description** |
| sort(l:List) | Sorts the list |
| sort(l:list, c:Compartor) | Sorts the list with the comparator |
| binarySearch(l:list,key:Object):int | Returns the location of key in the sorted list, negative otherwise |
| binarySearch(l:list,key:Object,c:Comparator):int | Returns the location of key in the sorted list using the comparator, negative otherwise |
| reverse(l:List) | Reverses the elements in the list |
| reverseOrder():Comparator | Returns a comparator with the reverse ordering |
| rotate(l:List,distance:int) | Rotates the elements in the list by distance |
| shuffle(l:List) | Shuffles the elements in the list randomly |
| copy(dest:List,source:List) | Copies source list into destination list |
| nCopies(n:int,o:Object):List | Returns a list with *n* copies of the object |
| fill(l:list,o:Object) | Fills the list with the object |
| max(c:Collection):Object | Returns the max object in the collection |
| max(c:Collection,cmp:Comparator):Object | Returns the max object in the collection using the comparator |
| min(c:Collection):Object | Returns the min object in the collection |
| min(c:Collection,cmp:Comparator):Object | Returns the min object in the collection using the comparator |
| disjoint(c1:Collection,c2:Collection):Boolean | Return true if c1 and c2 have no elements in common |
| frequency(c:Collection,o:Object):int | Returns the number of occurrences of the object in the collection. |

The *reverseOrder* method above is also overloaded to take a *Comparator*. For example, we could reverse the order of the SSN comparator:

List<Employee> employees = new LinkedList<>();

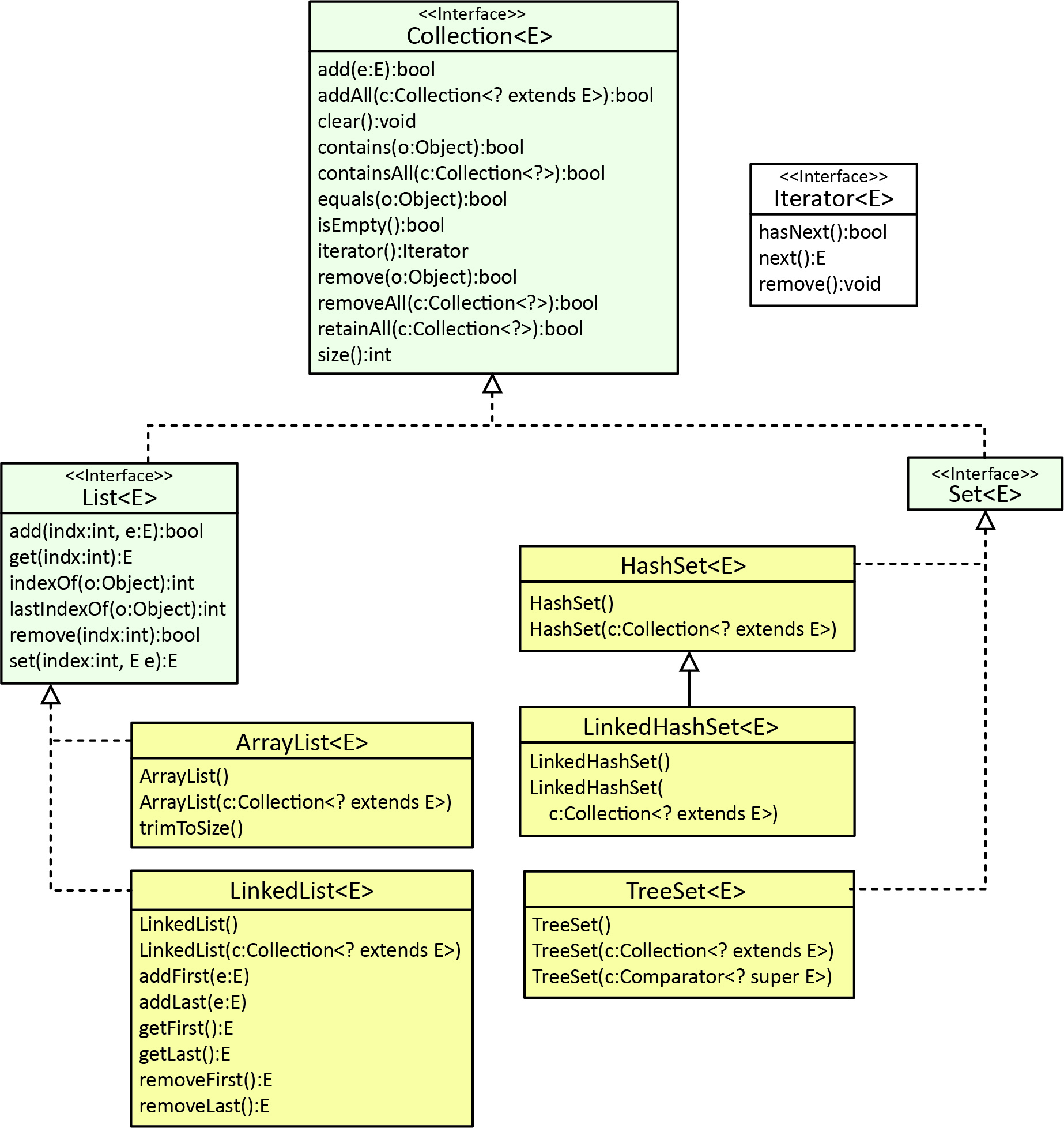
...

EmployeeSSNComparator comp = new EmployeeSSNComparator();

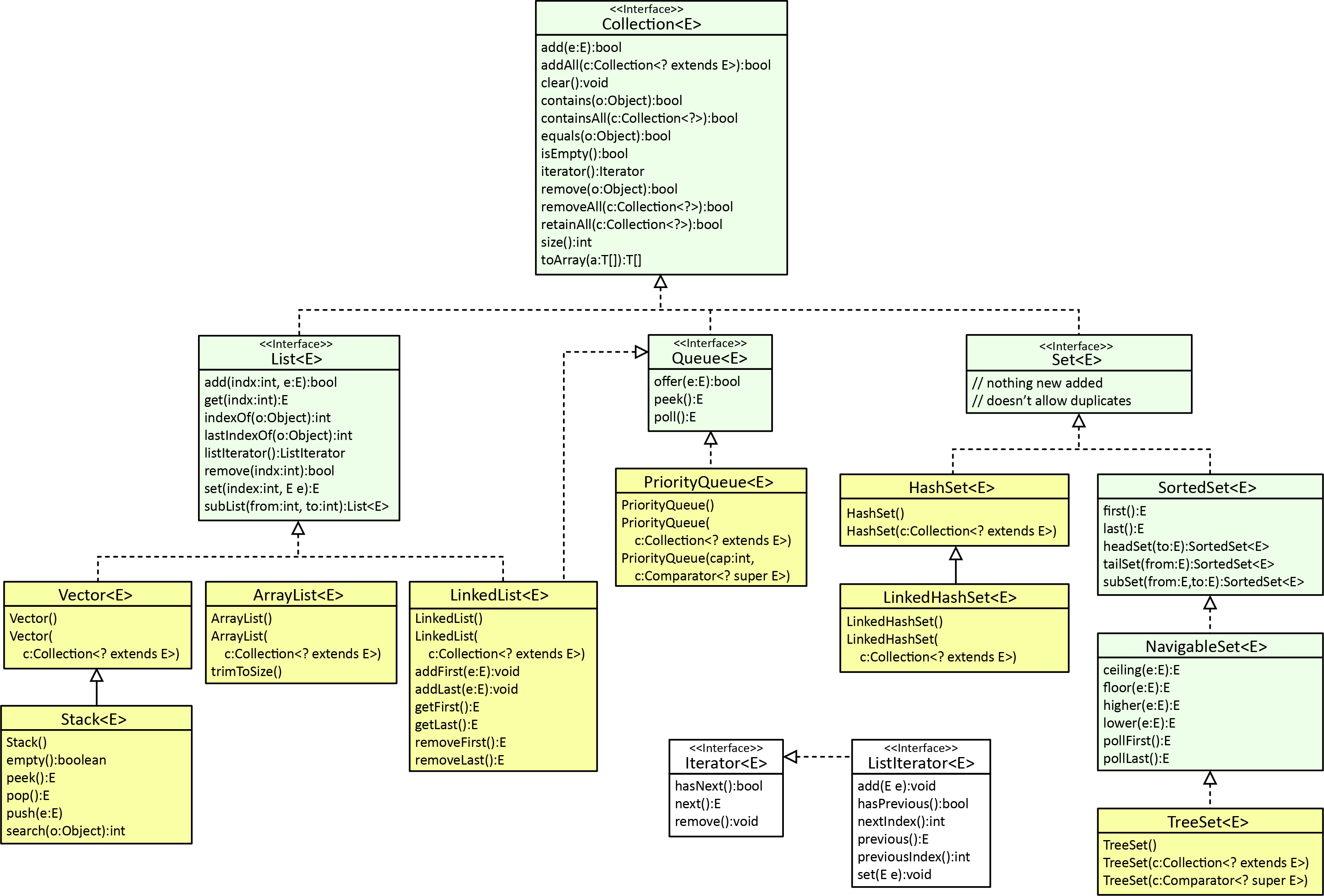
Collections.sort(employees, Collections.reverseOrder(comp));

1. Class Diagram for Collection Classes

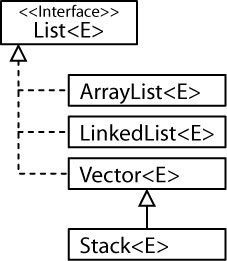
These are the classes and methods you are responsible for on a test. I will provide a copy of this image on a test.



This is a fuller version of the collection classes and methods.

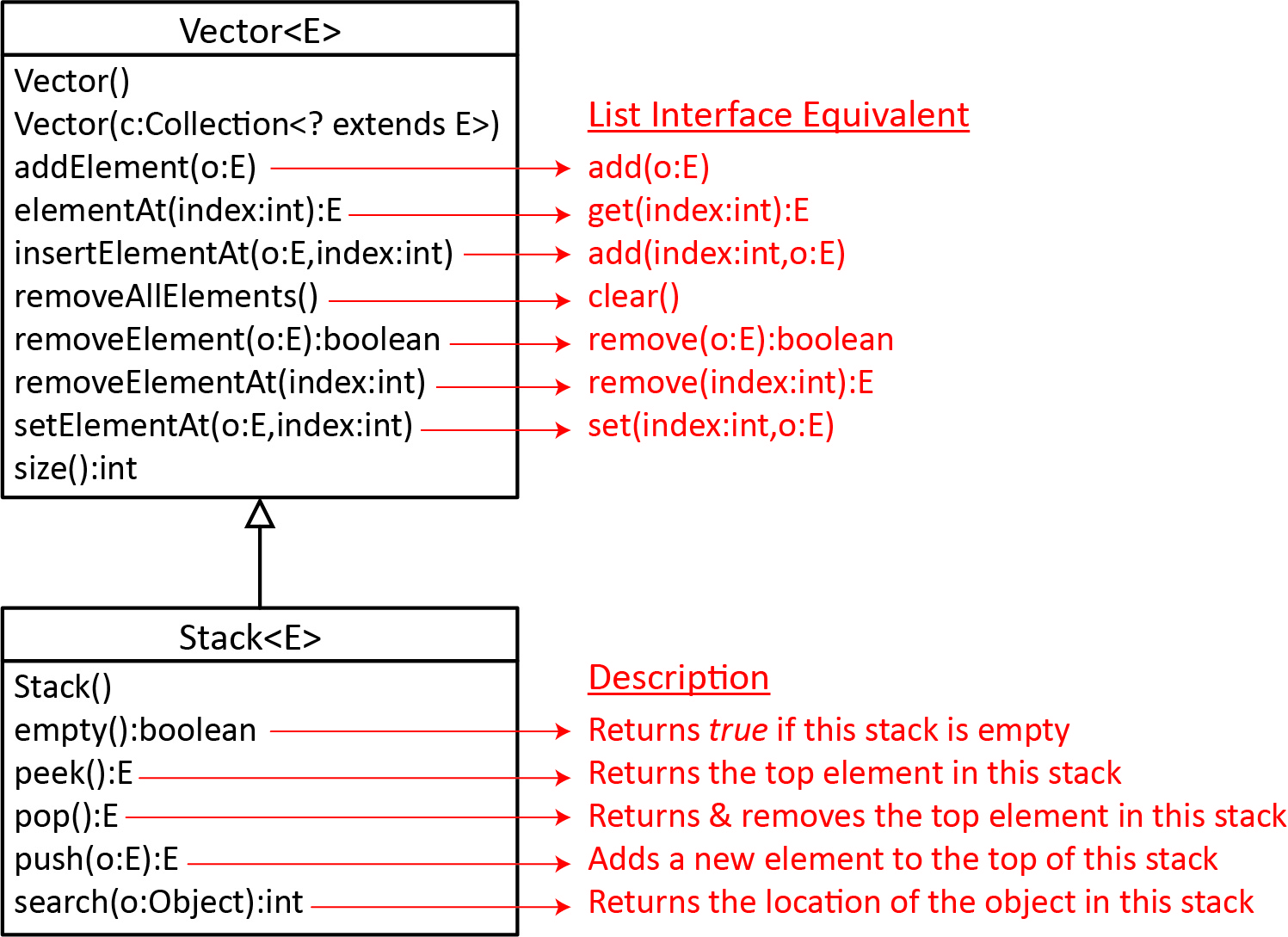


1. The Vector & Stack Classes

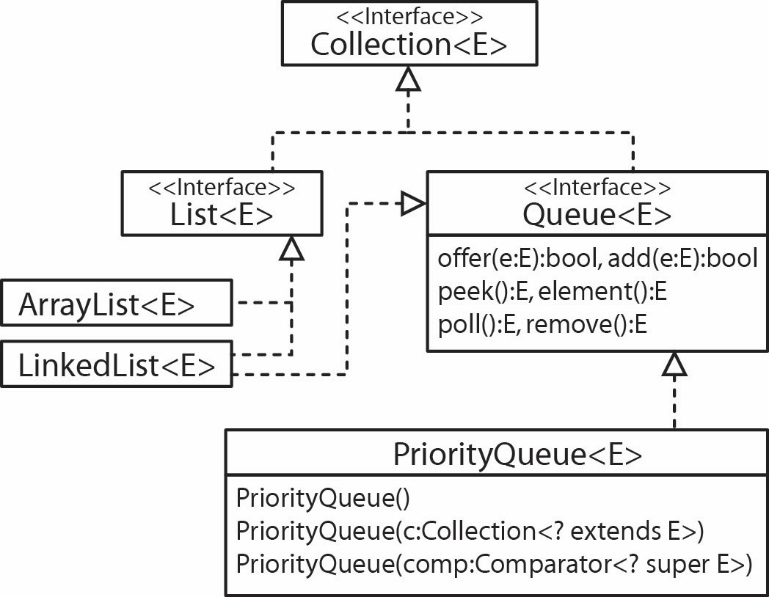
The Java Collections Framework (JCF) was introduced with Java 2 (1998). Thus, prior to that there was no *ArrayList, HashSet,* etc. The earliest version of Java supported two data structures: the *Vector* class and the *Stack* class. As shown in the diagram on the right, they were retrofitted to be generic and to fit in the JCF. However, the method names were preserved to support compatibility.

*Vector* is the same as *ArrayList*, except (a) that *Vector* contains the synchronized methods for accessing and modifying the elements. Synchronization means that the class is *thread-safe*, *i.e.* multiple threads can use the structure without stepping on each other, (b) it has legacy methods, *e.g. insertElement, etc.*  Note that *ArrayList* and *LinkedList* are not synchronized. If synchronization is required, you can use the synchronized versions of the collection classes which can be obtained with static methods in the *Collections* class (*e.g. synchronizedList*).

The Vector and Stack classes are shown in the class diagram below.



1. Queues and Priority Queues

A *Queue* is a structure that models things that are *waiting*, for instance people waiting in line, or jobs waiting to be processed by a processor. When you add (*offer*) an element to a queue it is placed at the end (*tail*), when you remove (*poll*) an element from a queue it is removed from the front (*head*). A queue is usually a *first-in first-out (FIFO)* data structure. As shown in the class diagram on the right, the *Queue* interface extends the *Collection* interface. The three methods in *Queue* are:

|  |  |
| --- | --- |
| Methods | Description |
| offer(e:E):bool,  add(e:E):bool | Adds *e* to the end (*tail*) of the queue. *offer* returns *false* if the element cannot be added while *add* throws an exception. |
| peek():E,  element():E | Retrieves but doesn’t remove the first element (*head*) of the queue. *peek* returns *null* if the queue is empty while *element* throws and exception. |
| poll():E,  remove():E | Retrieves and removes the first element (*head*) in the queue. *poll* returns *null* if the queue is empty while *remove* throws an exception. |

Java does not define a concrete *Queue* class (*Queue* is an interface). However, as the class diagram on the right above shows, a *LinkedList* is-a *Queue*. For example, we could create a queue like this:

Queue<Integer> ints = **new** LinkedList<>();

Java defines a concrete *PriorityQueue* class as shown in the diagram above which implements *Queue.* A *PriorityQueue* orders its elements according to their natural ordering using *Comparable* or a *Comparator.* The element with the *least* value is assigned the *highest priority* and is the first one removed. In other words, when you *offer* an element to a *PriorityQueue,* the lower the value the closer it’s location is to the head of the queue.

For example:

Queue<Integer> ints = **new** PriorityQueue<>();

ints.offer(8);

ints.offer(2);

ints.offer(5);

**while**(ints.size()>0) {

System.***out***.print(ints.poll() + " ");

}

Output: 2 5 8

Example – We create a *PriorityQueue* of *Employee* objects. The constructor requires a *Comparator* (and also an initial capacity, we won’t discuss this, below, we just use the value 20).

Employee e1 = **new** Employee("Boggs", "Kay", 716533892, 12.57);

Employee e2 = **new** Employee("Lyton", "Ben", 476227851, 77.88);

Employee e3 = **new** Employee("Boggs", "Amy", 553572246, 22.32);

Employee e4 = **new** Employee("Dern", "Donald", 243558673, 23.44);

// Create priority queue to hold Employee objects

Queue<Employee> pqEmps = **new** PriorityQueue<>(20, **new** EmployeeSalaryComparator() );

// Add employees to p.queue

pqEmps.offer(e1);

pqEmps.offer(e2);

pqEmps.offer(e3);

pqEmps.offer(e4);

**while**( pqEmps.size() > 0) {

// Remove employee from head of queue and print

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

}

Output:

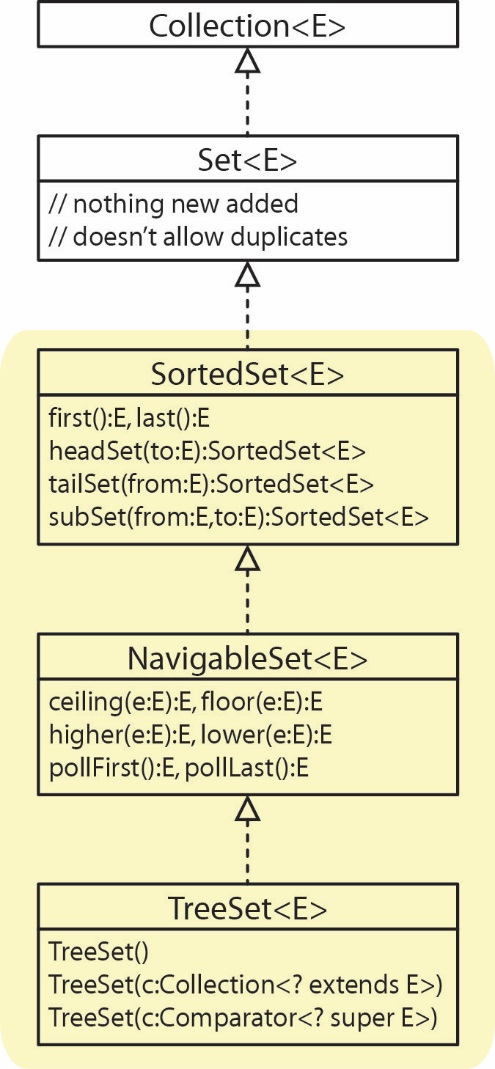
(Boggs, Kay - 716533892, 12.57)

(Boggs, Amy - 553572246, 22.32)

(Dern, Donald - 243558673, 23.44)

(Lyton, Ben - 476227851, 77.88)

1. The *SortedSet* & *NavigableSet* Interfaces

Above, we showed that *TreeSet* is-a *Set.* Actually, there are two interfaces in between (as shown in the class diagram on the right) that prescribe methods that provide access to certain elements.

|  |  |
| --- | --- |
| **Method** | **Description** |
| first():E | The first (smallest) element is returned |
| last():E | The last (largest) element is returned |
| headSet(to:E)  :SortedSet<E> | Returns a *SortedSet* of elements that are strictly less than *toElement*. {x|x<toElement} |
| tailSet(from:E)  :SortedSet<E> | Returns a *SortedSet* of elements greater than or equal to *fromElement*. {x|x>=fromElement} |
| subSet(from:E,to:E)  :SortedSet<E> | Returns a *SortedSet* of elements between *fromElement,* inclusive to *toElement* exclusive.  {x|fromElement <= x < toElement} |

See Lab 10 for examples of these methods.

Note that *headSet, tailSet, subSet* return a *SortedSet*. This is an odd structure. Consider the documentation for the *headSet* method:

*Returns a view of the portion of this set whose elements are strictly less than toElement. The returned set is backed by this set, so changes in the returned set are reflected in this set, and vice-versa.*

Thus, there may be situations where you might want to create a *TreeSet* from the *SortedSet* in order to break this bond. In other words, you want to preserve the result of *headSet* and then subsequently modify the *TreeSet* or *vice-versa*.

Next, we consider a few of the methods specified in the *NavigableSet* interface. The first four below) are similar to the methods in *SortedSet* except that they return a single item (or nothing).

|  |  |
| --- | --- |
| **Method** | **Description** |
| floor(e:E) | The largest element <= e is returned |
| lower(e:E) | The largest element < e is returned |
| ceiling(e:E) | The smallest element >= e is returned |
| higher(e:E) | The smallest element > e is returned |
| pollFirst() | Returns the smallest element and removes it |
| pollLast() | Returns the largest element and removes it |
| \*headSet(to:E,in:bool):NavigableSet<E> | Returns elements {x|x<=to} , when *in=true* |
| \*tailSet(from:E,in:bool):NavigableSet<E> | Returns elements {x|x>=from} , when *in=true* |
| \*subSet(to:E,in1:bool,from:E,in2:bool)  :NavigableSet<E> | Returns elements {x|from<=x<=to} , when *in1=true* and *in2=true* |
| \*descendingIterator():Iterator<E> | Returns an iterator over the elements in this set, in descending order. |
| \*descendingSet():NavigableSet<E> | Returns a reverse order view of the elements contained in this set. |

\* Not shown in class diagram above.

Similar to *SortedSet* above, *NavigableSet* is a view of the underlying set and changes to either are reflected in the other.

1. Actually, these methods are defined in the *Deque* interface. [↑](#footnote-ref-1)
2. <https://javarevisited.blogspot.com/2011/10/override-hashcode-in-java-example.html> [↑](#footnote-ref-2)
3. <https://medium.com/codelog/overriding-hashcode-method-effective-java-notes-723c1fedf51c> [↑](#footnote-ref-3)