**CS 1302 – Lab 12**

This is a tutorial about *Set*s: *HashSet* and *TreeSet*. There are 6 stages to complete this lab:

|  |  |  |
| --- | --- | --- |
| **Stage** | **Title** | **Text Reference** |
| 1 | The *HashSet* Class | 21.1-21.2 |
| 2 | *HashSet* Example – Unique Words | 21.4 |
| 3 | *HashSet* Example – Set Operations | 21.4 |
| 4 | Speed Comparison: *HashSet*, *ArrayList* & *LinkedList* | 21.3 |
| 5 | The *TreeSet* Class | 21.2 |
| 6 | *TreeSet* with Custom Objects | 21.2 |

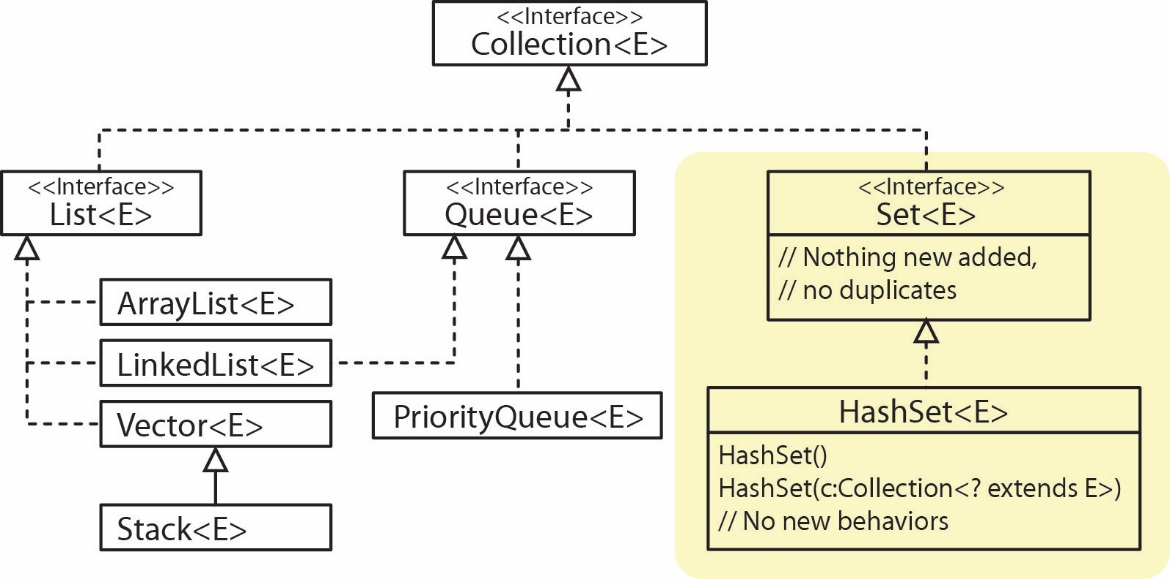
To make this document easier to read, it is recommend that you turn off spell checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”
3. **The *HashSet* Class**

In this stage we consider the *Set* interface and the *HashSet* class.

1. **Read (no action required)** –
2. As shown in the class diagram below, *Set* is a subinterface of *Collection.* The *Set* interface

* Introduces no new methods (other than what is in *Collection*)
* Does not allow duplicate elements. More specifically, a set can contain no two elemenets, *e1* and *e2* such that *e1.equals(e2)* is *true*.



1. The *HashSet* class implements the *Set* interface and introduces no new methods. Some important properties of a *HashSet*:

* The order of items is not preserved. If you add the integers 1, then 2 then 3 to a *HashSet* and then iterate over the elements, you might get them back out 2,3,1, or some other order.
* There is no sequential access; there is no *get* method. Remember that *get* is specified in the *List* interface and a *Set* is not a *List*. In other words, you can’t say “give me the 3rd item”.
* It has a constructor that accepts any type of *Collection*. Thus, you can turn any type of *Collection* into a *Set*, and a side affect is that any duplicate elements will be dropped.
* *HashSet* is the fastest of all collection classes at adding, removing, and seeing if an item is in the set (contains).

1. **Setup** – Do the following:
2. **Establish a Workspace** – Create a folder on your drive where you will put your lab or use an existing one.
3. **Run Eclipse** – As the program begins to run, it will ask you to navigate to the Workspace you want to use.
4. **Create a Project** – Create a Java project with the name, *lab12\_lastName*, *e.g. lab12\_gibson*.
5. Create a package named *hashset\_examples.*
6. **Run Example** – We illustrate the the order of insertion is not preserved in a *HashSet*.
7. Create a class named *HashSetExamples* (in the *hashset\_examples* package)and replace everything in the class (exceptthe package statement at the top) with:

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.HashSet;

**import** java.util.List;

**import** java.util.Scanner;

**import** java.util.Set;

**public** **class** HashSetExamples {

**public** **static** **void** main(String[] args) **throws** FileNotFoundException {

*hsExamples1*();

}

**public** **static** **void** hsExamples1() {

System.***out***.println("HashSet Example 1\n-----------------");

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

hsCities.add("Atlanta");

hsCities.add("New York");

hsCities.add("Durango");

hsCities.add("New York"); // duplicate, will not be added

System.***out***.println(" Order cities added to set: Atlanta, New York, Durango");

System.***out***.print("Order when set iterated with for loop: ");

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

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

}

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

}

}

1. Run the code and verify the output. Your objective is to see how a *HashSet* is created, how to add items, that duplicates are not added, how to iterate over them, and to illustrate that the order that items are added is not preserved.
2. **Run Example** – In this example we show how we can remove the duplicate items in an *ArrayList* by creating a *HashSet* from it.
3. **Read (no action required)** – *HashSet* has a constructor that takes any type of *Collection* (*List, Queue, Set, etc*) adding the elements of that *Collection* to the *HashSet*. Since *HashSet* doesn’t allow duplicates, any duplicates will be ignored. For example:

List<String> words = **new** ArrayList<>(Arrays.*asList*("not", "go", "at", "not", "go"));

Set<String> uniqueWords = **new** HashSet<String>(words);

1. Add the method below to the *HashSetExamples* class:

**public** **static** **void** hsExamples2() {

System.***out***.println("\nHashSet Example 2\n-----------------");

List<String> words = **new** ArrayList<>(Arrays.*asList*("not", "go", "at", "see", "go", "be", "not"));

System.***out***.println("Words in list: " + words);

Set<String> uniqueWords = **new** HashSet<String>(words);

System.***out***.println("Words in set : " + uniqueWords);

}

1. Add the line of code below to the end of *main* so that the method is called.

*hsExamples2*();

1. Run the code and verify the output.
2. ***HashSet* Example – Unique Words**

In this stage we consider a problem where a *HashSets* is useful.

1. **Read (no action required)** –
2. We consider an example where we will read a text file that contains words (for simplicity the file will only contain words, no punctuation nor other markup). For example:

The dog is a rabbit until it is not a dog

1. From these words we will create two lists: one that contains all the words in the file (a single occurrence of each), and another list that contains duplicate words (words that occur more than once). For example, from the input above we will produce (remember, the order is not preserved in a *HashSet*):

All words: [The, a, not, rabbit, is, until, it, dog]

Duplicate words: [a, is, dog]

We will do this using two *HashSets*, *allWords* and *duplicateWords*.

1. First, we will read a word:

String word = input.next();

1. Next, we attempt to add the word to *uniqueWords*. If the word is unique, then it will be added. The *add* method returns *true* if the word was added and *false* otherwise. Thus, if the return is *false* (*i.e.* the word is not added) then we will simply add the word to the *duplicateWords* set. Of course if the word already exists in *duplicateWords,* it will not be added. For example:

**if**(!allWords.add(word)) {

duplicateWords.add(word);

}

Note that we could have written the block above, perhaps slightly more understandable, as shown below:

**if**(!allWords.contains(word)) {

allWords.add(word);

}

**else** **if**(!duplicateWords.contains(word)) {

duplicateWords.add(word);

}

1. **Create Text File**
2. Select the *hashset\_examples* package in the Solution Explorer.
3. Choose: File, New, Untitled Text File.
4. Copy the line of text below into the file:

The dog is a rabbit until it is not a dog

1. Choose: File, Save As
2. Navigate to the *src/hashset\_examples* folder.
3. Supply the *File name*: “words.txt”
4. Choose: OK
5. Verify that *words.txt* is in the *hashset­\_examples* package. Drag it there if necessary.
6. **Run Example** – We code the preceding example
7. Add the method below to the *HashSetExamples* class:

**private** **static** **void** hsExamples3() **throws** FileNotFoundException {

System.***out***.println("\nHashSet Example 3\n-----------------");

String path = "src/hashset\_examples/words.txt";

Scanner input = **new** Scanner(**new** File(path));

HashSet<String> allWords = **new** HashSet<>();

HashSet<String> duplicateWords = **new** HashSet<>();

// Reads file and creates a HashSet of all words in the text and

// a HashSet of words that are duplicates.

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

String word = input.next();

**if**(!allWords.add(word)) {

duplicateWords.add(word);

}

}

input.close();

System.***out***.println(" All words: " + allWords);

System.***out***.println("Duplicate words: " + duplicateWords);

}

1. Add the line of code below to the end of *main* so that the method is called.

*hsExamples3*();

1. Run the code and verify the output.
2. ***HashSet* Example – Set Operations**

In this stage we consider another problem we can solve with *HashSets*.

1. **Read (no action required)** –
2. Consider the following situation, we have a set of employees (represented by their integer id) that have completed training module 1:

Set<Integer> hsEmps1 = **new** HashSet<>(Arrays.*asList*(1,2,3,4,5,6,7));

And, we have another set that have completed training module 2:

Set<Integer> hsEmps2 = **new** HashSet<>(Arrays.*asList*(1,3,5,6,7,8,9));

Thus, among other observations, we can see that employee 1 has completed both training modules, employee 2 has completed only the first module, and employee 8 has completed only the second module.

1. Now, suppose we want to write code to create a newset that has the employees that have completed both training modules. We can do this by first creating a new *HashSet* with all the employees that have completed training module 1:

Set<Integer> hsTeamsInCommon = **new** HashSet<>(hsEmps1);

1. Next, we intersect (*retainAll*) this new set with the set of employees that have completed the second module:

hsTeamsInCommon.retainAll(hsEmps2);

Thus, *hsTeamsInCommon* contains all the employees that have completed both training modules and that we haven’t modified either of the original sets: *hsEmps1* and *hsEmps2.*

1. In the example that follows, we write code to create sets that contain all employees (ids) who complete: (a) both modules, (b) either module (1,2, or both), (c) only module 1, (d) only module 2, (e) exactly one of the modules. For example:

Employees who completed:

module 1: [1, 2, 3, 4, 5, 6, 7]

module 2: [1, 3, 5, 6, 7, 8, 9]

Employees who completed both:

[1, 3, 5, 6, 7]

Employees who completed either module:

[1, 2, 3, 4, 5, 6, 7, 8, 9]

Employees who completed only module 1:

[2, 4]

Employees who completed only module 2:

[8, 9]

Employees who had exactly 1 module:

[2, 4, 8, 9]

Note: Observe that the order of insertion appears to be preserved in these sets. This is conincidence.[[1]](#footnote-2)

1. Similarly, we can create the following sets:

// Employees who completed either training module (1,2, or both):

Set<Integer> hsEmpsEither = **new** HashSet<>(hsEmps1);

hsEmpsEither.addAll(hsEmps2);

// Employees who completed only module 1 (but not module 2)

Set<Integer> hsOnlyMod1 = **new** HashSet<>(hsEmps1);

hsOnlyMod1.removeAll(hsEmps2);

// Employees who completed only module 2 (but not module 1)

Set<Integer> hsOnlyMod2 = **new** HashSet<>(hsEmps2);

hsOnlyMod2.removeAll(hsEmps1);

// Employees who completed exactly one module (either 1 or 2, but not both)

Set<Integer> hsExactly1Mod = **new** HashSet<>(hsOnlyMod1);

hsExactly1Mod.addAll(hsOnlyMod2);

1. **Run Example** – We code the preceding example
2. Add the method below to the *HashSetExamples* class:

**private** **static** **void** hsExamples4() {

System.***out***.println("\nHashSet Example 4\n-----------------");

// Employees who completed training module 1

Set<Integer> hsEmps1 = **new** HashSet<>(Arrays.*asList*(1,2,3,4,5,6,7));

// Employees who completed training module 2

Set<Integer> hsEmps2 = **new** HashSet<>(Arrays.*asList*(1,3,5,6,7,8,9));

// Employees who completed both training modules.

Set<Integer> hsTeamsInCommon = **new** HashSet<>(hsEmps1);

hsTeamsInCommon.retainAll(hsEmps2);

System.***out***.println("Employees who completed:\n " + "module 1: " + hsEmps1 + "\n " + "module 2: " + hsEmps2);

System.***out***.println("Employees who completed both:\n " + hsTeamsInCommon);

// Employees who completed either training module (1,2, or both):

Set<Integer> hsEmpsEither = **new** HashSet<>(hsEmps1);

hsEmpsEither.addAll(hsEmps2);

System.***out***.println("Employees who completed either module:\n " + hsEmpsEither);

// Employees who completed only module 1 (but not module 2)

Set<Integer> hsOnlyMod1 = **new** HashSet<>(hsEmps1);

hsOnlyMod1.removeAll(hsEmps2);

System.***out***.println("Employees who completed only module 1:\n " + hsOnlyMod1);

// Employees who completed only module 2 (but not module 1)

Set<Integer> hsOnlyMod2 = **new** HashSet<>(hsEmps2);

hsOnlyMod2.removeAll(hsEmps1);

System.***out***.println("Employees who completed only module 2:\n " + hsOnlyMod2);

// Employees who completed exactly one module (either 1 or 2, but not both)

Set<Integer> hsExactly1Mod = **new** HashSet<>(hsOnlyMod1);

hsExactly1Mod.addAll(hsOnlyMod2);

System.***out***.println("Employees completed exactly 1 module:\n " + hsExactly1Mod);

}

1. Add the line of code below to the end of *main* so that the method is called.

*hsExamples4*();

1. Run the code and verify the output.
2. **Speed Comparison: *HashSet, ArrayList & LinkedList***

In this stage we will do an experiment to compare how much time it takes to remove values from *HashSet, ArrayList,* and *LinkedList*.

1. **Read (no action required)** – This is similar to the speed comparison we did in Lab 9.
2. Consider the following experiment: Suppose we have a *HashSet* that initially contains 50,000 (unique) random integers. We will time how long it takes to remove 25,000 random integers from this *HashSet*. Note that almost none of the removes will be successful (*i.e.* they don’t exist in *HashSet,* however this does not change the dramatic time results we will see). Here is an algorithm:

Create a HashSet, *hashSet* with 50,000 (unique) random integers.

Create an ArrayList, *vals* with 25,000 (unique) random integers.

Start timing

For(int x : vals)

hashSet.remove(x)

Stop timing

1. Next, consider repeating this experiment where the *HashSet* (*hashSet*)initially has 100,000 random integers (we will still remove just 25,000). Then 150,000.
2. Repeat all the above except using a *LinkedList*, and then repeat using *ArrayList.* The results I obtains are pretty dramatic. Note, the time to do the removes for:

* *LinkedList* and *ArrayList* takes increasingly longer time as the size of the collection increases.
* *HashSet* does not seem to be affected by the size of the collection and is much faster.

1. **Run Example**
2. Create a class named *SpeedComparison* (in the *hashset\_examples* package)and replace everything in the class (exceptthe package statement at the top) with:

**import** java.util.ArrayList;

**import** java.util.Collection;

**import** java.util.Collections;

**import** java.util.HashSet;

**import** java.util.LinkedList;

**import** java.util.List;

**import** java.util.Set;

**public** **class** SpeedComparison {

**static** **final** **int**[] ***INITIAL\_LIST\_SIZE*** = {50000, 100000, 150000, 200000};

**static** **final** **int** ***NUM\_REMOVES*** = 25000;

**public** **static** **void** main(String[] args) {

**for**( **int** listSize : ***INITIAL\_LIST\_SIZE***) {

// Create ArrayList

ArrayList<Integer> aryList = *generateArrayList*(listSize);

// Create values to remove

ArrayList<Integer> valsToRemove = *generateRemovesList*(***NUM\_REMOVES***,aryList);

// Create LinkedList from ArrayList

List<Integer> lnkList = **new** LinkedList<>(aryList);

// Create HashSet from ArrayList

Set<Integer> hashSet = **new** HashSet<>(aryList);

*doExperiment*(lnkList, valsToRemove);

*doExperiment*(aryList, valsToRemove);

*doExperiment*(hashSet, valsToRemove);

}

}

**public** **static** ArrayList<Integer> generateArrayList(**int** numValues) {

ArrayList<Integer> ints = **new** ArrayList<>();

**int** numAdded = 0;

**while**( numAdded < numValues ) {

// Generate an integer between 0 and max int

**int** val = (**int**)(Math.*random*()\*Integer.***MAX\_VALUE***);

**if**( !ints.contains(val)) {

// If value is not in set, then add it to set and list

ints.add(val);

numAdded++;

}

}

**return** ints;

}

**public** **static** ArrayList<Integer> generateRemovesList(**int** numValues, ArrayList<Integer> vals) {

// Build set of indices

List<Integer> indices = **new** ArrayList<>();

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

indices.add(i);

}

// So that random order is achieved

Collections.*shuffle*(indices);

// Build removes list

ArrayList<Integer> removes = **new** ArrayList<>();

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

removes.add(vals.get(indices.get(i)));

}

**return** removes;

}

**public** **static** **void** doExperiment(Collection<Integer> list, Collection<Integer> vals) {

// Use for output

**int** initialSize = list.size();

String className = list.getClass().getName();

**int** locPeriod = className.lastIndexOf(".");

className = className.substring(locPeriod+1);

String msg = String.*format*("%s size: %d, time to remove %d vals: ", className, list.size(), vals.size());

// Begin timing

**long** begTime = System.*currentTimeMillis*();

**for**(**int** x : vals) {

list.remove(x);

}

// Stop timing

**long** endTime = System.*currentTimeMillis*();

// Calculate total time in seconds.

**double** totTime = (endTime-begTime)/1000.0;

msg += String.*format*("%.3f sec", totTime);

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

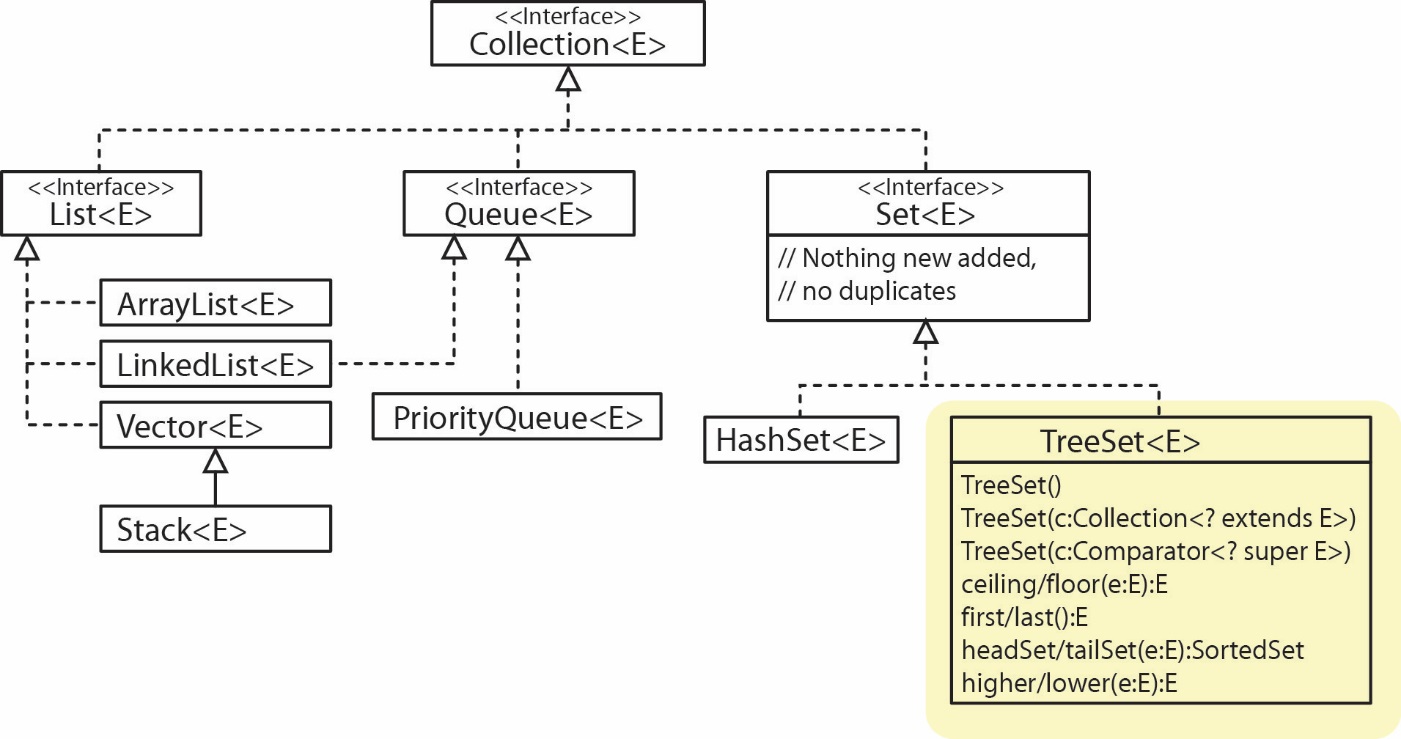
}

}

1. Run the code. It will probably take close to a minute to finish. Study the results, they probably will be astounding! Notice that the *ArrayList* and *LinkedList* take longer and longer; however, the *HashSet* doesn’t increase in time (or hardly at all, it may even decrease!) even as the size of the set gets larger.
2. **The *TreeSet* Class**

In ths stage we consider the *TreeSet* class.

1. **Read (no action required)** – *TreeSet* is another implementation of the *Set* interface as shown in the class diagram below.



Some important properties of a *TreeSet*:

* Items are ordered from smallest to largest according to their natural ordering or a *Comparator.*
* There is no sequential access; there is no *get* method. However, it does introduce a number of methods to return the *first* (smallest), *last* (largest), and various subsets (*headset, tailSet, subSet*).
* If you want to create a *TreeSet* of custom objects, the class must either implement *Comparable* or a *Comparator* must be supplied in the constructor.
* *TreeSet* is the fastest of all collection classes at keeping items ordered.

1. **Run Example**
2. Create a package named: *treeset\_examples*.
3. Create a class named *TreeSetExamples* (in the *treeset \_examples* package)and replace everything in the class (exceptthe package statement at the top) with:

**import** java.util.Set;

**import** java.util.SortedSet;

**import** java.util.TreeSet;

**public** **class** TreeSetExamples {

**public** **static** **void** main(String[] args) {

*tsExamples1*();

}

**public** **static** **void** tsExamples1() {

System.***out***.println("TreeSet Example 1\n-----------------");

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

tsCities.add("New York");

tsCities.add("Atlanta");

tsCities.add("Durango");

tsCities.add("Moab");

System.***out***.println(" Order cities added: New York, Atlanta, Durango, Moab");

System.***out***.print("Access cities (ordered) with for loop: ");

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

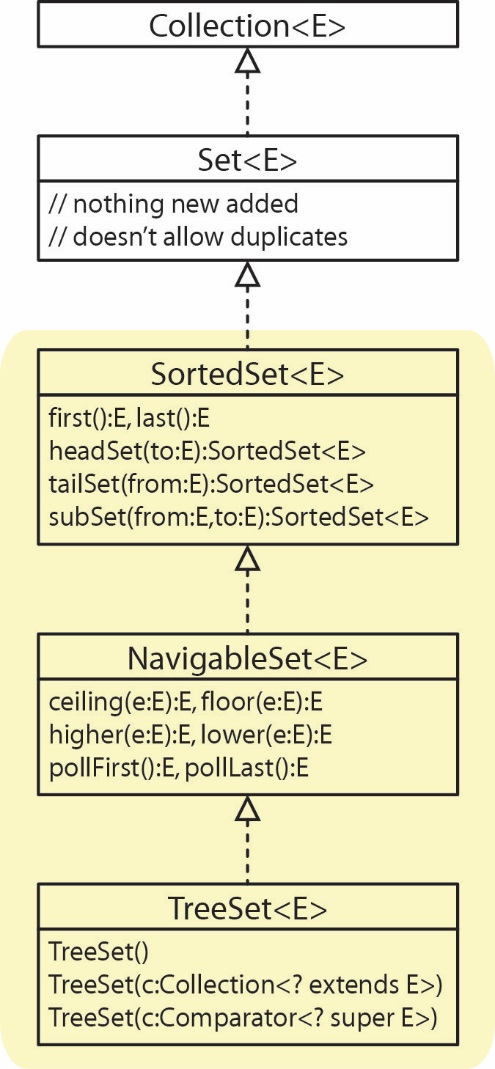
System.***out***.print(city + ", ");

}

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

}

}

1. Run and verify the output. Your objective to observe that a *TreeSet* keeps the elements ordered.
2. **Read (no action required)** –
3. 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 the behaviors we saw in *TreeSet*  . We first consider a few of the methods specified in the *SortedSet* interface:

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

1. Note: The later three methods have overloaded versions that allow a Boolean to specify whether to be inclusive or exclusive with respect to the “fromElement” and/or “toElement”.
2. **Run Example**
3. Add the method below to the *TreeSetExamples* class:

**public** **static** **void** tsExamples2() {

System.***out***.println("\nTreeSet Example 2\n-----------------");

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

tsCities.add("New York");

tsCities.add("Atlanta");

tsCities.add("Durango");

tsCities.add("Moab");

tsCities.add("Chicago");

System.***out***.print("Access cities (ordered) with for loop: ");

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

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

}

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

System.***out***.println("\nfirst(): " + tsCities.first());

System.***out***.println("last(): " + tsCities.last());

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

SortedSet<String> ssHead = tsCities.headSet("Denver");

System.***out***.println("Cities less than 'Denver'");

System.***out***.println(" headSet(\"Denver\"): " + ssHead);

ssHead = tsCities.headSet("Durango");

System.***out***.println("Cities less than 'Durango'");

System.***out***.println(" headSet(\"Durango\"): " + ssHead);

ssHead = tsCities.headSet("Fort Worth");

System.***out***.println("Cities less than 'FortWorth'");

System.***out***.println(" headSet(\"FortWorth\"): " + ssHead);

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

SortedSet<String> ssTail = tsCities.tailSet("Denver");

System.***out***.println("Cities greater than 'Denver'");

System.***out***.println(" tailSet(\"Denver\"): " + ssTail);

ssTail = tsCities.tailSet("Durango");

System.***out***.println("Cities greater than 'Durango'");

System.***out***.println(" tailSet(\"Durango\"): " + ssTail);

ssTail = tsCities.tailSet("Fort Worth");

System.***out***.println("Cities greater than 'Fort Worth'");

System.***out***.println(" tailSet(\"Fort Worth\"): " + ssTail);

ssTail = tsCities.tailSet("Raleigh");

System.***out***.println("Cities greater than 'Raleigh'");

System.***out***.println(" tailSet(\"Raleigh\"): " + ssTail);

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

SortedSet<String> ssSub = tsCities.subSet("Chicago", "New York");

System.***out***.println("Cities between 'Chicago' (inclusive) and 'New York'");

System.***out***.println(" subSet(\"Chicago\", \"New York\"): " + ssSub);

ssSub = tsCities.subSet("A", "H");

System.***out***.println("Cities between 'A' (inclusive) and 'H'");

System.***out***.println(" subSet(\"A\", \"H\"): " + ssSub);

}

1. Add the line of code below to the end of *main* so that the method is called.

*tsExamples2*();

1. Run the code and verify the output.
2. **Read (no action required)** – Next, we consider a few of the methods specified in the *NavigableSet* interface. These (the first four below) are similar to the methods in *SortedSet* except that they return a single item (or nothing). We will not illustrate these nor show example code for these.

|  |  |
| --- | --- |
| **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 |

1. ***TreeSet* with Custom Objects**

In this stage we consider the *TreeSet* class that holds instances of custom class.

1. **Read (no action required)** –
2. In the example that follows, we consider a *TreeSet* that holds *Employee* objects ordered on *salary. TreeSet* has a constructor that accepts a *Comparator* to maintain the order of the set. Thus, we will use a *EmployeeSalaryComparator* (as considered in Lab 10)*.* For example:

TreeSet<Employee> tsEmps = **new** TreeSet<>(**new** EmployeeSalaryComparator());

tsEmps.add(**new** Employee("Orville", 553572246, 22.32));

...

1. Next, in the example we will find the set of *Employees* that have a salary 30 or higher. We will use an approach we used in Lab 10 where we created a “dummy” *Employee* object with just the information we know (the salary of 30.0). Then, we pass the dummy to the *tailSet* method:

Employee emp = **new** Employee("unknown", 0, 30.0);

SortedSet<Employee> sSet = tsEmps.tailSet(emp);

1. Finally, we illustrate a method from the *NavigableSet* interface, *floor,* to find the employee with the largest salary that is less than or equal to 30:

Employee eSal30 = tsEmps.floor(emp);

1. **Run Example**
2. Create a class named *Employee* (in the *treeset \_examples* package)and replace everything in the class (exceptthe package statement at the top) with:

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

**private** String name;

**private** **int** ssNum;

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

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

**this**.name = name;

**this**.ssNum = ssNum;

**this**.salary = salary;

}

**public** String getName() { **return** name; }

**public** **int** getSSNum() { **return** ssNum; }

**public** **double** getSalary() { **return** salary; }

**public** String toString() {

**return** String.*format*("Name: %-8s - SSN: %d\tSalary: $%.2f", getName(),

getSSNum(), getSalary() );

}

}

1. Create a class named *EmployeeSalaryComparator* (in the *treeset \_examples* package)and replace everything in the class (exceptthe package statement at the top) with:

**import** java.util.Comparator;

**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;

}

}

1. Create a class named *EmployeeDriver* (in the *treeset \_examples* package)and replace everything in the class (exceptthe package statement at the top) with:

**import** java.util.Set;

**import** java.util.SortedSet;

**import** java.util.TreeSet;

**public** **class** EmployeeDriver {

**public** **static** **void** main(String[] args)

{

TreeSet<Employee> tsEmps = **new** TreeSet<>(**new** EmployeeSalaryComparator());

tsEmps.add(**new** Employee("Orville", 553572246, 22.32));

tsEmps.add(**new** Employee("Boggs", 716533892, 12.57));

tsEmps.add(**new** Employee("Lyton", 476227851, 77.88));

tsEmps.add(**new** Employee("Dern", 243558673, 23.44));

tsEmps.add(**new** Employee("Abscome", 994334662, 55.23));

System.***out***.println("Original List");

*printList*(tsEmps);

// Get employees with Salary 30 or higher. First, create "dummy" employee.

Employee emp = **new** Employee("unknown", 0, 30.0);

SortedSet<Employee> sSet = tsEmps.tailSet(emp);

System.***out***.println("\nEmployees with Salary >= 30: ");

*printList*(sSet);

// Get employee with largest Salary <= 30. Use "dummy" employee from above.

Employee eSal30 = tsEmps.floor(emp);

System.***out***.println("\nEmployee with largest Salary <= 30: " + eSal30);

}

**private** **static** **void** printList(Set<Employee> emps) {

**for**(Employee e : emps) {

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

}

}

}

1. Run and verify the output.

**Submission**

1. Do the following:
2. Zip all the folders (packages) under the *src* folder into a zip file named: *lab12\_lastname.zip*
3. Upload your zip file to the *lab12* dropbox in Blazeview.

**You are done!**

1. <https://stackoverflow.com/questions/9345651/ordering-of-elements-in-java-hashset> [↑](#footnote-ref-2)