# Set Notes

A Set is an interface that extends a Collection. It is important to know that it is NOT a List. The two implementing classes that uses Set that we will be focusing on is the HashSet and the TreeSet. A Set is a collection that contains no duplicate elements. The HashSet holds all values in the order of the hash value, a value that is unique to every element created. The TreeSet holds all values in the order of the Comparable the element is created under.

Create a file called ExampleSets and write the following:

**import** java.util.Set;

**import** java.util.TreeSet;

**import** java.util.HashSet;

**import** java.util.\*;

**import** **static** java.lang.System.\*;

**public** **class** ExampleSets {

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

{

TreeSet <Integer> set = **new** TreeSet<Integer>();

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

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

//Set<Double> money = new Set<Double>();

}

}

## Instantiating a Set

Since a Set is an interface, it has no constructor to instantiate. However HashSet and TreeSets are classes and do have constructors. The programmer can always instantiate a HashSet and a TreeSet in the normal generic way. However, if the programmer wants to create a set reference as a general set, then the programmer still must instantiate that set as either a HashSet or a TreeSet. If you uncomment the last line in the current example, you will get a compile-time error because there is no Set constructor.

In this class, we will often use TreeSet because we know what the order of the Set would look like, we would use HashSet less often, but a HashSet is quicker because each element has its own identity and can be found immediately in the Set, giving it a Big O run time of O(1). A Hash Set uses a Hash table to store its value and so it goes directly to the value it is looking for.

Add the following to the code

HashSet <Integer> digits = **new** HashSet<Integer>();

set.add(43);

set.add(32);

set.add(35);

set.add(48);

set.add(43);

set.add(35);

set.add(4);

set.add(3);

**for**(**int** i=0; i<10; i++)

digits.add(i);

***out***.println("Set:" + set);

***out***.println("Digits:" + digits);

The output should look like this:

Set:[3, 4, 32, 35, 43, 48]

Digits:[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## The add method

The add method will add the element to the set. The programmer tried to input several numbers into the set that already existed, yet the program did not crash. That is because the add method is a Boolean return method and it returns true if the element was added to the set and false otherwise. If the element is already in the set, the method just returns false. The example also shows that toString has been overridden to return everything in the set as a String. The digits set came out in order, but that is only because the hashValue of the 10 digits happen to be in the same order as the compareTo values of the elements.

Add the following to the code:

digits.addAll(set);

***out***.println("Digits:"+digits);

***out***.println("Set contains 48?"+set.contains(48));

***out***.println("Set contains -7?"+set.contains(-7));

digits.clear();

digits.add(3);

digits.add(4);

***out***.println("Set contains all digits?"+set.containsAll(digits));

digits.add(-7);

***out***.println("Set contains all digits?"+set.containsAll(digits));

set.addAll(digits);

***out***.println("Set contains all digits?"+set.containsAll(digits));

***out***.println("Set equals digits?"+set.equals(digits));

***out***.println("Set is empty?"+set.isEmpty());

***out***.println("Names is empty?"+names.isEmpty());

The output should look like this

Digits:[0, 32, 1, 2, 3, 35, 4, 5, 6, 7, 8, 9, 43, 48]

Set contains 48?true

Set contains -7?false

Set contains all digits?true

Set contains all digits?false

Set contains all digits?true

Set equals digits?false

Set is empty?false

Names is empty?true

addAll will place the parameter Collection into the Set object calling the method. contains will return true if the set contains the object being passed it, otherwise it will return false. containsAll will return if the Set contains all of the Collection being passed and, if so, will return true. Otherwise containsAll will return false. The Collection being set does not have to be a Set as it is in the exampled, it could also be a List. equals returns true if both the object and the parameter have the same value, false otherwise. This means that the Object that equals receives in the parameter must be a Set. isEmpty returns true when the Set contains no values and false otherwise.

Add the following to the code:

Iterator <Integer> iter = set.iterator();

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

names.add(""+iter.next());

***out***.println("Names:"+names);

***out***.print("Names:");

**for**(String k:names)

***out***.print(k+" ");

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

***out***.println("Set size:"+set.size());

//for(int i=0; i<set.size(); i++)

// out.println(set.get(i));

set.remove(35);

***out***.println("Set:"+set);

iter = set.iterator();

iter.next(); iter.next();

iter.remove(); iter.next();

***out***.println("Set after iter.remove():"+set);

set.remove(48);

//iter.next();

The output should look like this:

Names:[35, 3, 4, 48, -7, 32, 43]

Names:35 3 4 48 -7 32 43

Set size:7

Set:[-7, 3, 4, 32, 43, 48]

Set after iter.remove():[-7, 4, 32, 43, 48]

Sets are the main reason we learn iterator. In the example above, there are three ways to go to each and every element in an array. The first way is to set up an iterator and use a while loop to iterate to each element. The program goes to every Integer in set and adds those values to names, effectively making them Strings. Also, notice that when names printed out, they were printed in hash order and not in Comparable order, because names is a HashSet. The second way of going through a set is to use the enhanced for loop, which written to behave like an iterator. That is why you cannot have concurrent modifications when you use an enhanced for loop. The third way cannot work and that is why it is commented out. The program tried to call an index for each element in the Set but the Set is not a List and therefore does not have any indices. Notice that there is a method in Set that gives the size of the Set called size().

The program then used the remove method on the Set interface. This will go directly to the Object to be removed. If that object is there, then it will remove the object and return true, otherwise it does nothing and returns false. The program then shows the iterator and how it would remove a value from the Set. Notice that there is an iter.next() that is commented out. That is because if you use the remove method from Set, then the iterator must be reset because the Set has changed independently of the iterator.

Add the following to the code:

names.clear();

names.add("Giants");names.add("Cardinals");

names.add("Federals");names.add("Cowboys");

names.add("Eagles");

words.add("Dodgers"); words.add("Giants");

words.add("Padres"); words.add("Cardinals");

words.add("Cubs");

***out***.println("names:"+names);

***out***.println("words:"+words);

names.retainAll(words);

***out***.println("names:"+names);

words.removeAll(names);

***out***.println("words:"+words);

The output should look like this:

names:[Cardinals, Giants, Cowboys, Federals, Eagles]

words:[Cardinals, Cubs, Dodgers, Giants, Padres]

names:[Cardinals, Giants]

words:[Cubs, Dodgers, Padres]

The sets names and words were filled with words to show how removeAll and retainAll works. removeAll will receive a Collection as a parameter and remove all the elements from that collection from the set object calling the method. retainAll will receive a Collection as a parameter and retain only the elements from that collection in the set calling the method. The Collection could be a list in either case, but the Set will never have duplicates in the set.