AbstractSet<E> class

AbstractSet<E> class is abstract layer for Set<E> interface. Previously we saw few abstract classes like [AbstractCollection<E>](http://data-structure-learning.blogspot.com/2015/05/java-collection-part-10.html), [AbstractList<E>](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-11-abstractlist.html), [AbstractSequentialList<E>](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-12.html), [AbstractQueue<E>](http://data-structure-learning.blogspot.com/2015/05/abstractqueue-class.html).

Implementing Set<E> interface means that there are constraints imposed by Set<E> interface.

AbstractSet<E> class definition

AbstractSet<E> is an abstract class which extends AbstractCollection<E> and implements Set<E> interface.

**public** **abstract** **class** AbstractSet<E> **extends** AbstractCollection<E> **implements** Set<E>

Constructor for AbstractSet<E> class

Constructor of this class is protected.

**protected** AbstractSet() {

}

equals(Object o)

This method is used to compare the Object passed in parameter. This method returns true only if, both are set, both sets have same size & every member are contained in given set. Below is the code with comments in it. Read the comments as it will be helpful in understanding of internal working of equals(..) method.

**public** **boolean** equals(Object o) {

/\*\*

\* If passed object is of "this" instance

\* return true.

\* \*/

**if** (o == **this**)

**return** **true**;

/\*\*

\* If passed object is not instance of Set

\* return false

\* \*/

**if** (!(o **instanceof** Set))

**return** **false**;

//Case the passed object to Collection<?>

Collection<?> c = (Collection<?>) o;

/\*\*

\* If size of both of then is different

\* return false.

\*/

**if** (c.size() != size())

**return** **false**;

**try** {

/\*\*

\* containsAll(c) method of AbstractCollection<E> is called

\* returns true of it contains all element in specified collection

\* in for this collection.

\*

\* We have compared size before so this is will yield proper

\* output.

\* \*/

**return** containsAll(c);

} **catch** (ClassCastException unused) {

**return** **false**;

} **catch** (NullPointerException unused) {

**return** **false**;

}

}

hashCode()

This method is used to return the hashCode() for this set. The hash code is defined as sum of all the hash codes of elements in the set. If the element is null the hash code is 0. So it implies that s1.equals(s2) has s1.hashCode() == s2.hashCode()

**public** **int** hashCode() {

**int** h = 0;

//iterate throught the elements and add the hashCode for every element.

Iterator<E> i = iterator();

**while** (i.hasNext()) {

E obj = i.next();

**if** (obj != **null**)

h += obj.hashCode();

}

**return** h;

}

removeAll()

Removes all the elements from this set that are in specified collection. If the specified collection is also the set then the value remaining after this operation is asymmetric set difference of 2 sets.

This implementation first checks the size of the specified collection and this set.

If **size of collection smaller** will have its own iterator, check the element returned by iterator see if it exists, If element exists then it is removed. For this scenario remove(..) method of set is used.

Else **if the set size is small** then we take iterator of set and check if collection has element returned by iterator. If yes then remove the element by iterator’s remove method.

**public** **boolean** removeAll(Collection<?> c) {

Objects.*requireNonNull*(c);

**boolean** modified = **false**;

**if** (size() > c.size()) {

**for** (Iterator<?> i = c.iterator(); i.hasNext(); )

modified |= remove(i.next());

} **else** {

**for** (Iterator<?> i = iterator(); i.hasNext(); ) {

**if** (c.contains(i.next())) {

i.remove();

modified = **true**;

}

}

}

**return** modified;

}

That’s all on AbstractSet<E> class.