AbstractCollection<E> class

AbstractCollection<E> class is in top hierarchy or provides skeletal implementation of [Collection<E>](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-4collection.html) interface. To learn about Collection<E> interface click [here](http://data-structure-learning.blogspot.com/2015/05/java-collections-part-4collection.html).

Let us start understanding AbstractCollection<E> class and its methods. This class is abstract so few methods does not have concrete implementation as those methods are specific to implementing classes.

**Code for this class is referred and take from Javadocs.**

**AbstractCollection<E> class definition**

AbstractionCollection <E> is abstract class that implements Collection<E> interface. It is abstract as it does not implement all the methods.

**public** **abstract** **class** AbstractCollection<E> **implements** Collection<E>

**Constructor for AbstractCollection<E> class**

Constructor of this class is protected by definition. And why is that? The reason is that the invocation can be done by subclass constructors only.

**protected** AbstractCollection() {

}

**iterator()**

The iterator for the Collection is abstract as we cannot determine at this level how will we iterate through the collection.

**public** **abstract** Iterator<E> iterator();

**size()**

size() is also an abstract method as we do not know how many elements are inserted into collection.

**public** **abstract** **int** size();

**isEmpty()**

This method returns whether Collection is empty or not.

**public** **boolean** isEmpty() {

**return** size() == 0;

}

**contains(Object o)**

This me iterates over the collection and checks whether the object passed as parameter to contains method exists or not in collection.

/\*\*

\* Checks for the Object equality for parameter Object o

\* and object in Collection<E>

\* \*/

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

//Get the iterator to iterate on Collection<E>

Iterator<E> it = iterator();

//If input parameter is null then check for null elements

**if** (o == **null**) {

**while** (it.hasNext())

//If null elements exists then return true.

**if** (it.next() == **null**)

**return** **true**;

}

/\*\*

\* If input parameter is not null then check for Object

\* equality using equals(Object) method of Object class.

\* If equal then return true.

\* \*/

**else** {

**while** (it.hasNext())

**if** (o.equals(it.next()))

**return** **true**;

}

/\*\*

\* Collection is iterated but no match found

\* return false.

\* \*/

**return** **false**;

}

**toArray methods**

|  |  |
| --- | --- |
| Method | Description |
| **public** Object[] toArray() | This method returns the Object[] |
| **public** <T> T[] toArray(T[] a) | This method returns the T[] specified in the parameter. |

**add(E)**

add(E) method throws UnsupportedOperationException() because the class with concrete implementation must implement this method. This method is not responsibility of AbstractCollection<E>.

**public** **boolean** add(E e) {

**throw** **new** UnsupportedOperationException();

}

**remove(Object o)**

This method iterates over the collection and searched for Object passed as parameter in collection. If found then it removed the element by using Iterator’s own remove method.

/\*\*

\* Checks for the Object equality for parameter Object o

\* and object in Collection<E>.

\* If found it will be removed.

\* return true if element is deleted.

\* return false if element is not found.

\* \*/

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

//Get the iterator to iterate on Collection<E>

Iterator<E> it = iterator();

//If input parameter is null then check for null elements

**if** (o == **null**) {

**while** (it.hasNext()) {

/\*\*

\* If null elements exists then remove it

\* return true once element is removed.

\*/

**if** (it.next() == **null**) {

it.remove();

**return** **true**;

}

}

}

/\*\*

\* If input parameter is not null then check for Object

\* equality using equals(Object) method of Object class.

\* If equal then remove it.

\* return true once element is removed.

\* \*/

**else** {

**while** (it.hasNext()) {

**if** (o.equals(it.next())) {

it.remove();

**return** **true**;

}

}

}

/\*\*

\* Collection is iterated but no match found

\* return false.

\* \*/

**return** **false**;

}

**containsAll(Collection<?> c)**

This method iterates over the collection and checks whether each element in collection is in specified collection. If all are present then return true, else return false.

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

**for** (Object e : c)

**if** (!contains(e))

**return** **false**;

**return** **true**;

}

**addAll(Collection<? Extends E> c)**

This implementation iterates over specified collection and adds each object returned by iterator to this collection. Returns true if collection is modified, else return’s false.

**public** **boolean** addAll(Collection<? **extends** E> c) {

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

**for** (E e : c)

**if** (add(e))

modified = **true**;

**return** modified;

}

**removeAll(Collection<?> c)**

This implementation iterates over specified collection and checks for each object returned by iterator to this collection. If element exists then remove from this collection with iterator’s method.

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

Objects.*requireNonNull*(c);

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

Iterator<?> it = iterator();

**while** (it.hasNext()) {

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

it.remove();

modified = **true**;

}

}

**return** modified;

}

**retainAll(Collection<?> c)**

This implementation iterates over specified collection and checks for each object returned by iterator to this collection. If it is not contained then it will remove that element from specified collection.

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

Objects.*requireNonNull*(c);

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

Iterator<E> it = iterator();

**while** (it.hasNext()) {

**if** (!c.contains(it.next())) {

it.remove();

modified = **true**;

}

}

**return** modified;

}

**clear()**

This implementation iterates over this collection removing each element by iterator’s remove() method.

**public** **void** clear() {

Iterator<E> it = iterator();

**while** (it.hasNext()) {

it.next();

it.remove();

}

}

**toString()**

This method returns the string representation of the collection.

**public** String toString() {

Iterator<E> it = iterator();

**if** (! it.hasNext())

**return** "[]";

StringBuilder sb = **new** StringBuilder();

sb.append('[');

**for** (;;) {

E e = it.next();

sb.append(e == **this** ? "(this Collection)" : e);

**if** (! it.hasNext())

**return** sb.append(']').toString();

sb.append(',').append(' ');

}

}