Lets Code!

Collections - Iterator & Collection

Collections Framework

What we'll cover

What we'll cover

What is a collection?

What we'll cover

What is a collection?

The Collection Interface

What we'll cover

What is a collection?

The Collection Interface

The Iterator Interface

What we'll cover

What is a collection?

The Collection Interface

The Iterator Interface

The Iterable Interface

`Iterator` and `Collection` interface

What is a `Collection`?

- `Collection` is an interface which ensures a class has the ability to hold a series of objects.
- Often, we consider `Map` objects to be a `Collection`, although they do not *implement* the `Collection` interface.

Collections and Maps

List	Set	Map
ArrayList	HashSet	TreeMap
LinkedList	TreeSet	HashMap
ArrayDeque	PriorityQueue	WeakHashMap
	EnumSet	IdentityHashMap
	LinkedHashSet	LinkedHashMap

Collection Interface

- Fundamental interface for `Collection` classes in java.

```
public interface Collection<e> extends Iterable<e> {
    boolean add(E element);
    boolean addAll(Collection<!--? extends E--> collection);
    void clear();
    boolean contains(Object object);
    boolean containsAll(Collection<!--?--> collection);
    boolean isEmpty();
    Iterator<e> iterator();
    boolean remove(Object object);
    boolean removeAll(Collection<!--?--> collection);
    boolean retainAll(Collection<!--?--> collection);
    int size();
    Object[] toArray();
    <t> T[] toArray(T[] array);
}</t></e></e></e>
```

Collection Interface

`boolean add(E element)`

- Attempts to add an element to the `Collection`
- returns `true` if adding the element changes the `Collection`, else `false`.
- Adding an already-present-object to a `Set` collection will return `false`.

```
public void demo() {
    Collection<string> set = new HashSet<>();
    String valueToBeAdded = "Hedjet";
    set.add(valueToBeAdded);
    System.out.println(set.add(valueToBeAdded)); // prints false
}</string>
```

Collection Interface

`boolean addAll(Collection)`

- Attempts to add a collection of elements to the `Collection`
- returns `true` if adding the elements changes the `Collection`, else `false`.

```
public void demo() {
  Collection<string> set = new HashSet<>();
  String[] valuesToBeAdded = {"Froilan", "Tariq", "Eric", "Stephanie", "Leah"};
  Collection<string> valuesAsList = Arrays.asList(valueToBeAdded);
  System.out.println(set.addAll(list)); // prints true
}</string></string>
```

Collection Interface

`boolean remove(Object)`

- Attempts to remove an object from the `Collection`
- returns `true` if removing the element changes the `Collection`, else returns `false`
- Removing an element that is not present in an `ArrayList` will return `false`.

```java
public void demo() {
  // prints false
  System.out.println(new ArrayList().remove(new Object()));
}
```

Collection Interface

`boolean removeAll(Collection)`

- Attempts to remove a collection of elements from the `Collection`
- returns `true` if removing the elements changes the `Collection`, else returns `false`

```
public void demo() {
  String[] elementsAsArray = {"The", "Quick", "Brown"};
  Collection<string> originalCollection = new ArrayList<>();
  Collection<string> elementsAsList = Arrays.asList(elementsAsArray);

  // prints false
  System.out.println(originalCollection.removeAll(elementsAsList));
}</string></string>
```

Collection Interface

`boolean retainAll(Collection)`

- Retains only the elements in this collection that are contained in the specified collection.
- returns `true` if retaining the elements changes the `Collection`, else returns `false`

```
public void demo() {
  String[] originalArray = {"The", "Quick", "Brown"};
  String[] elementsToBeRetained = {"The", "Quick"};
  List<string> originalList = new ArrayList<>(Arrays.asList(originalArray));
  List<string> retentionList = Arrays.asList(elementsToBeRetained);

  // prints true
  System.out.println(originalList.retainAll(retentionList));
}</string></string>
```

Collection Interface

`boolean isEmpty()`

- returns `true` if the size of the `Collection` is `0`, else returns `false`.

```
public void demo() {
  String[] elementsAsArray = {"The", "Quick", "Brown"};
  Collection<string> elementsAsList = Arrays.asList(arrayOfStrings);
  System.out.println(elementsAsList.isEmpty()); // prints false
}</string>
```

Collection Interface

`int size()`

- Returns the number of elements in the `Collection`.

```
public void demo() {
  String[] elementsAsArray = {"The", "Quick", "Brown"};
  Collection<string> elementsAsList = Arrays.asList(arrayOfStrings);
  System.out.println(elementsAsList.size()); // prints 3
}</string>
```

Collection Interface

`void clear()`

- Removes all elements from the `Collection`.

```
public void demo() {
  String[] elementsAsArray = {"The", "Quick", "Brown"};
  List<string> elementsAsList = new ArrayList<>(Arrays.asList(arrayOfStrings));
  elementsAsList.clear();
  System.out.println(elementsAsList.isEmpty()); // prints true
}</string>
```

Collection Interface

`Object[] toArray()`

- Populates a new `Object[]` with the elements from this `Collection`

```
public void demo() {
  String[] elementsToAdd = {"The", "Quick", "Brown"};
  List<string> elementList = new ArrayList<>(Arrays.asList(elementsToAdd));
  Object[] listAsObjectArray = elementList.toArray();
}</string>
```

Collection Interface

`E[] toArray(E[])`

- Populates a new array of the *respective type* with the elements from this `Collection`

```
public void demo() {
  String[] elementsToAdd = {"The", "Quick", "Brown"};
  List<string> elementList = new ArrayList<>(Arrays.asList(elementsToAdd));

  int newArrayLength = elementList.size();
  String[] arrayToBePopulated = new String[newArrayLength];
  String[] listAsStringArray = elementList.toArray(arrayToBePopulated);
}</string>
```

Collection Interface

`Iterator<E> iterator()`

- Returns an object that implements the `Iterator` interface

Iterable Interface

- `Iterable` ensures the implementing class is a valid candidate for the `foreach` loop
- `Collection` extends `Iterable`, therefore all `Collection` types are valid candidates for the `foreach` loop.
- All `Iterables` must provide an implementation for `Iterator<E> iterator()`.
- Is **NOT** the same as the `Iterator` interface.

Iterable Interface

```
public interface Iterable<e> {
  Iterator<e> iterator();
  forEach(Consumer<!--? super E--> E);
}</e></e>
```

Iterator interface

- `Iterator` is used to visit the elements in the `Collection`, one by one.

```
public interface Iterator<e> {
  E next();
  boolean hasNext();
  void remove();
  default void forEachRemaining(Consumer<!--? super E--> action);
}</e>
```

Iterator Interface

- Repeatedly calling the `next()` method enables you to visit each element from the collection, one by one.
- `NoSuchElementException` is thrown upon invoking `next()` on an `Iterator` that has reached the end of the collection.
  - This can be prevented by evaluating the `hasNext()` method before calling `next()`.
  - The compiler translates `foreach` loops into a loop with an iterator.

```java
public static void printIterable(Iterable<object> iterable) {
    Iterator iterator = iterable.iterator();
    while(iterator.hasNext()) {
        System.out.println("Current Element = " + iterator.next());
    }
}</object>
```

Iterator Interface

- As of Java8, you can call the `forEachRemaining` method with a `Consumer` lambda expression.
- The lambda expression is invoked with each element of the iterator, until there are none left.

```
public static void printIterable(Iterable<object> iterable) {
  Iterator iterator = iterable.iterator();
  iterator.forEachRemaining((element) -> System.out.println(element));
}</object>
```

Iterator Interface

`next()`

- Think of Java iterators as being *between* elements.
- When you call **`next`**, the iterator jumps over the next element, and it returns a reference to the element that it just passed.

Iterator Interface

`remove()`

- removes the element that was returned by the last call to `next()`
- Often, you may need to view an element before deciding to delete it.
- It is illegal to call `remove()` if it wasn't preceded by a call to `next()`.

```
public void deleteFirstElement(Iterator<string> iterator) {
    iterator.next(); // skip first element
    iterator.remove(); // remove first element
}</string>
```

`AbstractCollection` Class

- The `Collection` interface declares 18 methods.
- To avoid implementing a lot of the fundamental methods, the `Collection` library developers created an `AbstractCollection` class.
- AbstractCollection has a concrete implementation of all `Collection` methods except `size()` and `iterator()`

Sample `AbstractCollection` implementation

```java
public class MyCollection<e> extends AbstractCollection<e>  {
    private final Iterable<e> iterable;
    public MyCollection(Iterable<e> iterable) {
        this.iterable = iterable;
    }

    @Override
    public Iterator<e> iterator() {  return iterable.iterator(); }

    @Override
    public int size() {
        List<e> list = new ArrayList<>();
        iterator().forEachRemaining(list::add);
        return list.size();
    }
}</e></e></e></e></e></e>
```