Agenda

- Collection FrameWork
- Traversal
- FailSafe and FailFast Iterator
- List
- ~~Queue~~

Collection Framework

- Collection framework is Library of reusable data structure classes that is used to develop application.
- Main purpose of collection framework is to manage data/objects in RAM efficiently.
- Collection framework was introduced in Java 1.2 and type-safe implementation is provided in 5.0 (using generics).
- Collection is available in java.util package.
- Java collection framework provides

1. Interfaces -- defines standard methods for the collections.
2. Implementations -- classes that implements various data stuctures.
3. Algorithms -- helper methods like searching, sorting, ...

Collection Hierarchy

- Interfaces: Iterable, Collection, List, Queue, Set, Map, Deque, SortedSet, SortedMap, ...
- Implementations: ArrayList, LinkedList, HashSet, HashMap, ...
- Algorithms: sort(), reverse(), max(), min(), ... -> in Collections class static methods

Collection interface

- Root interface in collection framework interface hierarchy.
- Most of collection classes are inherited from this interface (indirectly).
- Provides most basic/general functionality for any collection
- Abstract methods
  - boolean add(E e)
  - int size()
  - boolean isEmpty()
  - void clear()
  - boolean contains(Object o)
  - boolean remove(Object o)
  - boolean addAll(Collection<? extends E> c)
  - boolean containsAll(Collection<?> c)
  - boolean removeAll(Collection<?> c)
  - boolean retainAll(Collection<?> c)
  - Object[] toArray()
  - Iterator iterator() -- inherited from Iterable
- Default methods
  - default Stream stream()

  - default Stream parallelStream()
  - default boolean removeIf(Predicate<? super E> filter)

Iterable interface

- To traverse any collection it provides an Iterator.
- Enable use of for-each loop.
- In java.lang package
- Iterable yeilds an iterator
- Methods
  - Iterator iterator()
  - default Spliterator spliterator()
  - default void forEach(Consumer<? super T> action)

Iterator

- Part of collection framework (1.2)
- Methods
  - boolean hasNext()
  - E next()
  - void remove()

Enumeration

Since Java 1.0

- Methods
  - boolean hasMoreElements()
  - E nextElement()

Collections class

- Helper/utility class that provides several static helper methods
- Methods
  - List reverse(List list);
  - List shuffle(List list);
  - void sort(List list, Comparator cmp)
  - E max(Collection list, Comparator cmp);
  - E min(Collection list, Comparator cmp);
  - List synchronizedList(List list);

Collection vs Collections

1. Collection interface

- All methods are public and abstract. They implemented in sub-classes.
- Since all methods are non-static, must be called on object.

```
Collection<Integer> list = new ArrayList<>();
//List<Integer> list = new ArrayList<>();
//ArrayList<Integer> list = new ArrayList<>();
list.remove(new Integer(12));
```

2. Collections class

- Helper class that contains all static methods.
- We never create object of "Collections" class.

```
Collections.methodName(...);
```

Fail-fast vs Fail-safe Iterator

- If state of collection is modified (add/remove operation other than iterator methods) i.e structural change while traversing a collection using iterator and iterator methods fails (with ConcurrentModificationException), then iterator is said to be Fail-fast.
- The collections from java.util package have fail-fast iterators
- e.g. Iterators from ArrayList, LinkedList, Vector, ...
- If iterator allows to modify the underlying collection (add/remove operation other than iterator methods) while traversing a collection (NO ConcurrentModificationException), then iterator is said to be Fail-safe.
- The collections from java.util.concurrent package have fail-safe iterators.
- e.g. Iterators from CopyOnWriteArrayList, ...
- If any changes are done in the collection using these iterators then the changes may not be reflected using the same iterator however by creating the new iterator we can get the changes displayed.

Traversal

1. Using Iterator

```
  Iterator<Integer> itr = list.iterator();
  while(itr.hasNext()) {
      Integer i = itr.next();
      System.out.println(i);
  }
```

2. Using for-each loop

```
  for(Integer i:list)
      System.out.println(i);

// gets converted into Iterator traversal internally

for(Iterator<Integer> itr = list.iterator(); itr.hasNext();)  {
    Integer i = itr.next();
    System.out.println(i);
}
```

3. using for loop

```
for(int index=0; index<list.size(); index++) {
    Integer i = list.get(index);
        System.out.println(i);
}
```

4. Enumeration -- Traversing Vector (Java 1.0)

```
  // v is Vector<Integer>
  Enumeration<Integer> e = v.elements();
  while(e.hasMoreElements()) {
      Integer i = e.nextElement();
      System.out.println(i);
  }
```

- Enumeration behaves similar to fail-safe iterator.
- Since Java 1.0
- Methods
  - boolean hasMoreElements()
  - E nextElement()
- only useful when traversing with vector

List Interface

- Ordered/sequential collection.
- Implementations: ArrayList, Vector, Stack, LinkedList, etc.
- List can contain duplicate elements.
- List can contain multiple null elements.
- Elements can be accessed sequentially (bi-directional using Iterator) or randomly (index based).
- List enables searching in the list
- Abstract methods
  - void add(int index, E element)
  - String toString()
  - E get(int index)

  - E set(int index, E element)
  - int indexOf(Object o)
  - int lastIndexOf(Object o)
  - E remove(int index)
  - boolean addAll(int index, Collection<? extends E> c)
  - ListIterator listIterator()
  - ListIterator listIterator(int index)
  - List subList(int fromIndex, int toIndex)
- To store objects of user-defined types in the list, you must override equals() method for the objects.
- It is mandetory while searching operations like contains(), indexOf(), lastIndexOf().