Table of Contents

1 Collections Introduction

An array is an indexed collection of fixed number of homogeneous data elements.

Limitations of object arrays :

- Arrays are fixed in size, i.e, once we created an array there is no chance of increasing or decreasing size based on our requirement,Hence to use arrays compulsory we should know the size in advance, which may not be possible always.

- Arrays can hold only homegeneous data elements, ie. (Same type).
  Ex -

```
Student[] st = new Student[];
st[0] = new Student[];
st[1] = new Student[];
st[2] = new Customer[]; --- This is not allowed.
```

  But we can resolve this problem by using object type arrays.

```
Object[] obs = new Object[]
obs[0] = new Student[];
obs[1] = new Customer[];
```

- Arrays concept not built based on some data structure, hence readymade methods support is not available, for every requiremenet. Compulsory programmer is responsible to write the logic.

To resolve the above problem collection concept was introduced.

Advantages of collections over arrays

- Collections are growable in nature, hence based on our requiremenet we can increse or decrease the size.
- Collection can hold both homegeneous & hetrogenous objects.
- Every collection class is implemented based on some data structure concept. Hence readymade methods support is available.

Disadvantages of collections

- Performance point of view collections are not recommended to use this is the limitations of collections.

Difference between arrays and collections

Arrays	Collections
Arrays are fixed in size	Collections are growable in nature
Memory point of view arrays concepts are Not recommended	Memory point of view Collections concepts are higly recommended
Performance point of view arrays concepts are recommended	Performance point of view Collections concepts are Not recommended
Arrays can hold homogeneous data elements.	Collections can hold homogeneous & Heterogeneous data elements.
There is no underlying data structure, so readymade method support	There is a underlying data structure for each collection type, so readymade method support is available.
Arrays can be used to hold both primitives and objects.	Collections can be used to hold only objects but not premitives.

A collection allows a group of objects to be treated as a single unit.

Objects can be stored, retrieved, and manipulated as elements of a collection. Program design often requires the handling of collections of objects. The Java Collections Framework provides a set of standard utility classes for managing various kinds of collections.

The core framework is provided in the **java.util package and comprises three main parts:**

- The core interfaces that allow collections to be manipulated independently of their implementation. These generic interfaces define the common functionality exhibited by collections and facilitate data exchange between collections.
- A set of implementations that are specific implementations of the core interfaces, providing data structures that a program can readily use.
- An assortment of static utility methods found in the Collections and Arrays classes that can be used to perform various operations on collections and arrays, such as sorting and searching, or creating

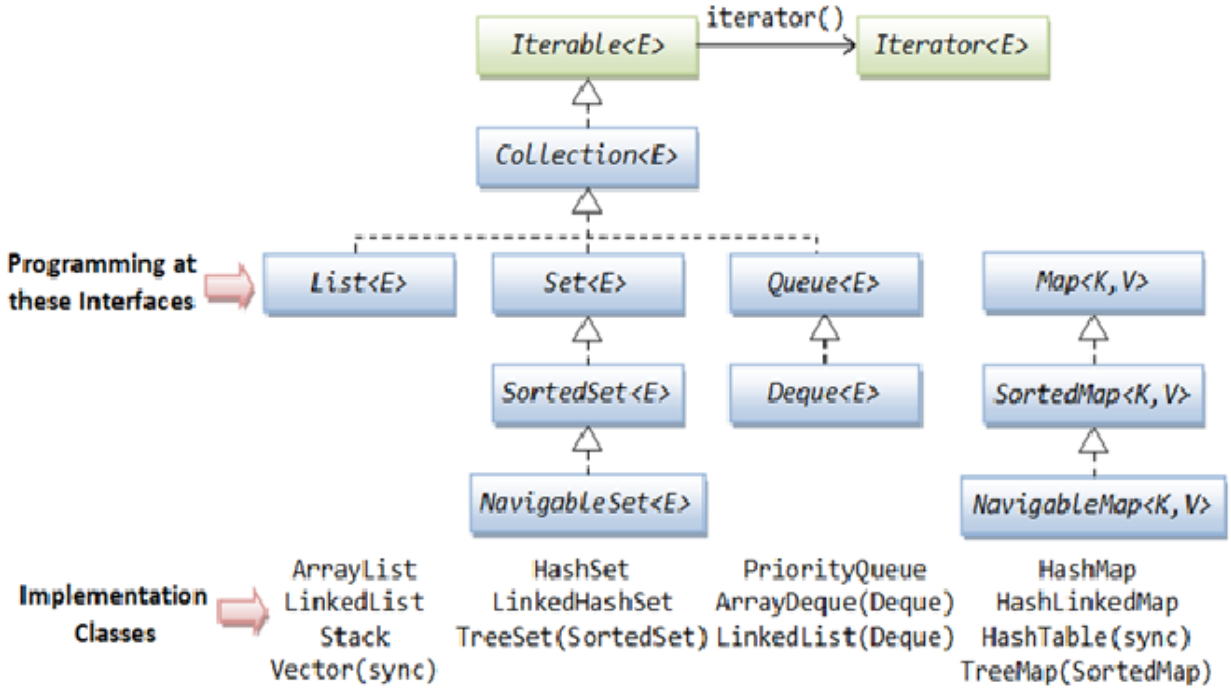
Collection Vs Collections

- Collection is an interface, can be used to represent a group of individual object as a single entity.
- Collections are a utility class, present in java.util. package, to define several utility methods for collections.

1.1 Core Interfaces

The generic Collection interface is a generalized interface for maintaining collections. The Collection interface extends the Iterable interface that specifies an iterator to sequentially access the elements of an Iterabl object, see the below diagram for quick reference of interface hierarchy.

Collection interface Hierarchy diagram:



List (Interface):

- It is the child interface of collection.
- If we want to represent a group of individual objects where insertion order is preserved and duplicates are allowed, then we should go for list.
- Vector & stack classes introduce in v1.0 version, and reengineered in v1.2 by introducing Arraylist and LinkedList.

Set (Interface) :

- It is the child interface of collection.
- If we want to represent a group of individual objects where duplicates are not allowed. then we should go for Set.

SortedSet (Interface) :

- It is the child interface of Set.
- If we want to represent a group of individual objects according to some sorting order then we should go got SortedSet.

NavigabledSet (Interface) :

- It is the child interface of SortedSet, to provide several methods for navigation purpose.
- It is introduced in version 1.6.

Queue (Interface) :

- It is the child interface of Collection,
- If we want to represent a group of individual objects prior to processing then we should go for Queue.

Map (Interface):

- If we want to represent group of Objects as key-value pair then go for Map.
- Both key and value are objects only.
- Duplicate keys are not allowed, but vlaues can be duplicated.
- Map is not a child interface of Collection.

SortedMap (Interface):

- If we want to represent group of Objects as key-value pair according to some sorting order then go for Map.
- Sorting should be done only based on keys, but not based on values.
- SortedMap is child Interface of Map.

NavigableMap (Interface):

- It is the child interface of SortedMap, to provide several methods for navigation purpose.
- It is introduced in version 1.6.

Cursor (Interface):

- Enumeration
- Iterator
- ListInterator

2 Collection Interface

Collection (Interface) -

If we want to represent a group of individual objects as a single entity then we should go for collection.

Collection interface defines the most common methods which can be applied for any collection objects.

The following is the list of methods present in collection interface

boolean add(Object 0)

boolean addAll(Collection 0) – Set operation **A ∪ B**

boolean remove(Object 0)

boolean removeAll(Collection 0) – Set operation **A-B**

boolean retainAll(Collection 0) - Set operation **A ∩ B**

void clear()

boolean isEmpty()

int size()

boolean contains(Object o)

boolean containsAll(Object o)

Object[] toArray()

Iterator iterator()

2.1 AbstractCollection

This class provides a skeletal implementation of the Collection interface, to minimize the effort required to implement this interface.

To **implement an un-modifiable collection**, the **programmer needs only to extend this class** and provide implementations for the **iterator and size methods**. (The iterator returned by the iterator method must implement hasNext and next.)

To implement a modifiable collection, the programmer must additionally override this class's add method (which otherwise throws an UnsupportedOperationException), and the iterator returned by the iterator method must additionally implement its remove method.

The programmer should generally provide a void (no argument) and Collection constructor, as per the recommendation in the Collection interface specification.

Below are the concrete methods:

Modifier and Type	Method and Description
boolean	**add**(**E** e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(**Object** o) Returns true if this collection contains the specified element.
boolean	**containsAll**(**Collection**<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.
boolean	**remove**(**Object** o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(**Collection**<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(**Collection**<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.
String	**toString**() Returns a string representation of this collection.

Below are the Abstract methods:

Modifier and Type	Method and Description
boolean	**add**(**E** e) Ensures that this collection contains the specified element (optional operation).
Boolean	**addAll**(**Collection**<? extends **E**> c) Adds all of the elements in the specified collection to this collection (optional operation).

3 List

List (Interface):

- It is the child interface of collection.
- If we want to represent a group of individual objects where insertion order is prese rved and duplicates are allowed, then we should go for list.
- Insertion order will be preserved by means of index.
- Vector & stack classes introduce in v1.0 version, and reengineered in v1.2 by introducing Arraylist and LinkedList.

List interface defines following methods

boolean add(o)

boolean add(int index, Object o)

boolean remove(int index)

Object get(int index)

Object set(int index, Object o)

int index(Object o)

int lastIndex(Object o)

ListIterator listIterator()

3.1 AbstractList

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).

To implement an unmodifiable list, the programmer needs only to extend this class and provide **implementations for the get(int) and size() methods.**

To implement a modifiable list, the programmer must additionally override the set(int, E) method (which otherwise throws an UnsupportedOperationException). If the list is variable-size the programmer must additionally override the add(int, E) and remove(int) methods.

Unlike the other abstract collection implementations, the programmer does not have to provide an iterator implementation; the iterator and list iterator are implemented by this class, on top of the "random access" methods: get(int), set(int, E), add(int, E) and remove(int).

3.2 ArrayList

Resizable-array implementation of the List interface.

In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.

- The size, isEmpty, get, set, iterator, and listIterator operations run in constant time.
- The add operation runs in amortized constant time, that is, adding n elements requires O(n) time.
- All of the other operations run in linear time, The constant factor is low compared to that for the LinkedList implementation.

Each ArrayList instance has a capacity (default 10). The capacity is the size of the array used to store the elements in the list. An **application can increase the capacity of an ArrayList** instance before adding a large number of elements using the **ensureCapacity operation.** This may reduce the amount of incremental reallocation.

This implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.

This is best done at creation time, to prevent accidental unsynchronized access to the list:

List list = Collections.synchronizedList(new ArrayList(...));

The iterators returned by this class's **iterator and listIterator methods are fail-fast**:

If the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

- The underlying datastructure is resizable array or growable array.
- Insertion order is preserved.
- Duplicate objects are allowed.
- Heterogeneous objects are allowed.
- Null insertion is allowed.
- Implements both RandomAccess, Clonable and Serializable interfaces.
- Best choice when frequent operations are retrievals, because of rand access behavior.
- Not a great choice if frequent operations are insertions at the middle of the ArrayList . Because it requires several shift operations.

Constructors :

1. ArrayList() - The default constructor creates a new, empty ArrayList with initial capacity 10. Once it reached max limit, a new arralist will be created with –

   newCapacity = currenctCapacity * 3/2 + 1

2. ArrayList(Collection<? extends E> c)

constructor creates a new ArrayList containing the elements in the specified collection. The new ArrayList will retain any duplicates. The ordering in the ArrayList will be determined by the traversal order of the iterator for the collection passed as argument.

3. ArrayList(int initialCapacity)
   constructor creates a new, empty ArrayList with the specified initial capacity.

Difference between Vectot and Arrraylist:

ArrayList	Vector
1) ArrayList is not synchronized.	Vector is synchronized.
2) ArrayList increments 50% of current array size if number of element exceeds from its capacity.	Vector increments 100% means doubles the array size if total number of element exceeds than its capacity.
3) ArrayList is not a legacy class, it is introduced in JDK 1.2.	Vector is a legacy class.
4) ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
5) ArrayList uses Iterator interface to traverse the elements.	Vector uses Enumeration interface to traverse the elements. But it can use Iterator also.

3.3 AbstractSequentialList

This class provides a skeletal implementation of the List interface to minimize the effort required to implement this **interface backed by a "sequential access" data store** (such as a linked list). For random access data (such as an array), AbstractList should be used in preference to this class.

To implement a sequential access list the programmer needs only to extend this class and provide implementations for the **listIterator and size methods.**

For an unmodifiable list, the programmer need only implement the list iterator's hasNext, next, hasPrevious, previous and index methods.

For a modifiable list the programmer should additionally implement the list iterator's set method. For a variable-size list the programmer should additionally implement the list iterator's remove and add methods.

3.4 LinkedList

- The underlying data structure is double Linked List.
- Insertion order is preserved.
- Duplicate objects are allowed.
- Heterogeneous objects are allowed.
- Null insertion is allowed.
- Implements both **List, Deque, Clonable and Serializable interfaces.**
- **Extends** AbstractSequentialList class.
- Best choice when frequent operations are insertions at the middle of list.
- Not a great choice if frequent operations are retrieval operations, as time complexity is O(n).
- All of the operations perform as could be expected for a doubly-linked list. Operations that index into the list will traverse the list from the beginning or the end, whichever is closer to the specified index.

Usually Linked list has below additional methods as its implementing Deque.

void addFirst(Object o)

void addLast(Object o)

Object removeFirst()

Object removeLast()

Object getFirst()

Object getLast()

Below are the difference between ArrayList and LinkedList.

ArrayList	Vector
1) ArrayList is **not synchronized**.	Vector is **synchronized**.
2) ArrayList **increments 50%** of current array size if number of element exceeds from its capacity.	Vector **increments 100%** means doubles the array size if total number of element exceeds than its capacity.
3) ArrayList is **not a legacy** class, it is introduced in JDK 1.2.	Vector is a **legacy** class.
4) ArrayList is **fast** because it is non-synchronized.	Vector is **slow** because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.

5) ArrayList uses **Iterator** interface to traverse the elements.	Vector uses **Enumeration** interface to traverse the elements. But it can use Iterator also.

3.5 Vector

- The underlying datastructure is resizable array or growable array.
- Insertion order is preserved.
- Duplicate objects are allowed.
- Heterogeneous objects are allowed.
- Null insertion is allowed.
- Implements RandomAccess, Clonable and Serializable interfaces.
- Best choice when frequent operations are retrievals, because of rand access behavior.
- Not a great choice if frequent operations are insertions at the middle of the ArrayList . Because it requires several shift operations.
- Every method in vector is synchronized. Hence Vector object is thread safe.

Vector specific methods -

boolean add(o) - C

boolean add(int index, Object o) -- L

boolean addElement(Object o) - V`

boolean remove(Object 0) - C

boolean remove(int index) - L

boolean removeElement(int index) - V

boolean removeElementAt(int index) - V

boolean removeAllElements() - V

clear()

Object get(int index)

Object getElementAt(int index)

Object firstElementAt()

Object lastElementAt()

Enumeration elements()

3.6 Stack

- It is the child class of vector contains only one constructor.
- The Stack class represents a last-in-first-out (LIFO) stack of objects.

- It extends class Vector with five operations that allow a vector to be treated as a stack. When a stack is first created, it contains no items.
- A more complete and consistent set of LIFO stack operations is provided by the Deque interface and its implementations, which should be used in preference to this class. For example:

  Deque<Integer> stack = new ArrayDeque<Integer>();

Constructors:

Stack s = new Stack()

Method :

Object push(Object o)

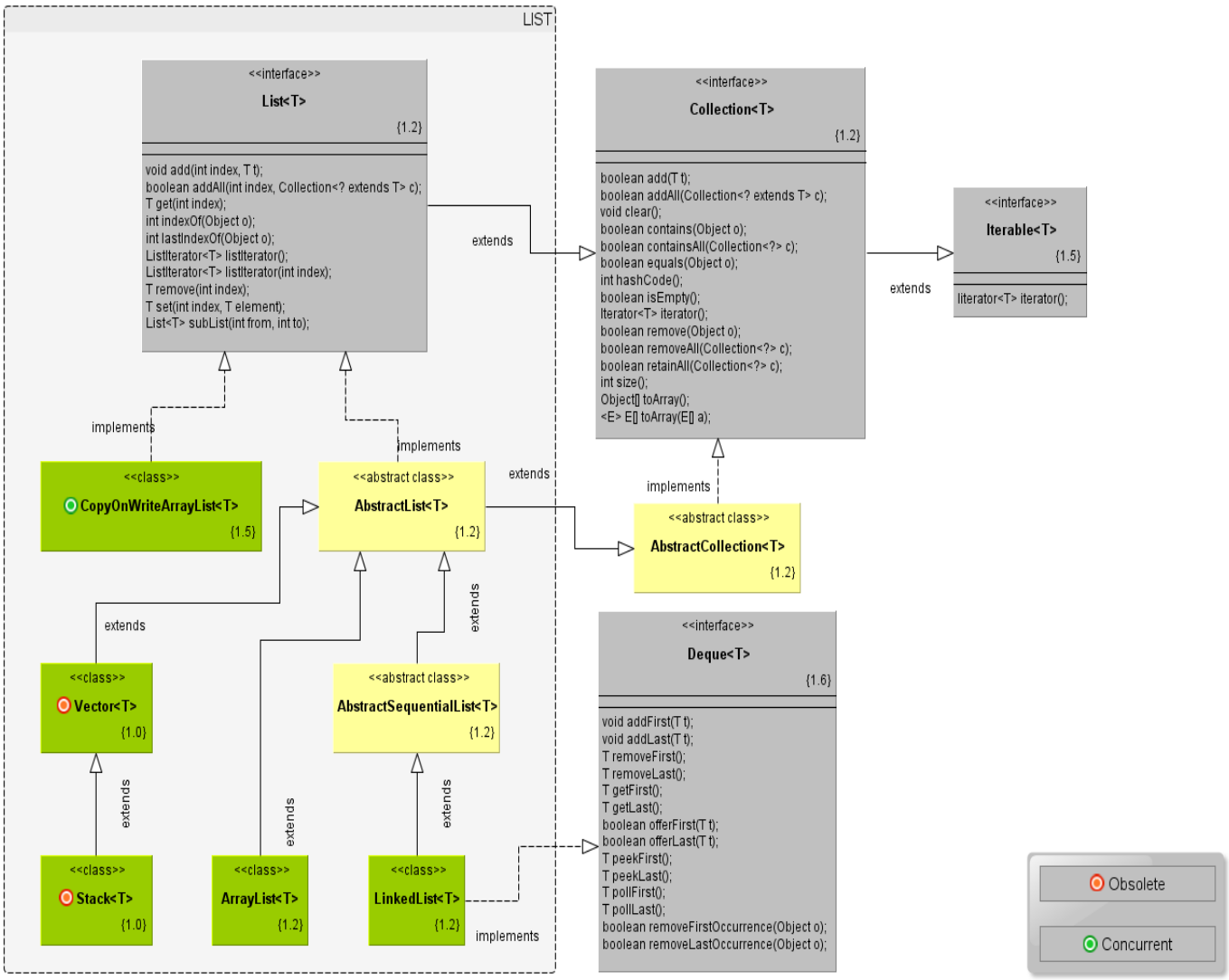
Object pop(Object o) - remove and return top of the stack

Object peek(Object o) - return top of the stack

boolean search(Object o) - returns offset from the top of the stack if the objectis available.

boolean empty() - returns true when stack is empty.

3.7 Class Diagram of List hierarchy



4 Collection Cursors

If we want to get object one by one from the collection we should go for cursor.

There are 3 cursors available in java

- Enumeration
- iterator
- ListIterator

4.1 Enumeration

It is a cursor to retrieve objects one by one from the collection.

- It is applicable for legacy classes.
- we can create enumeration object by using elements() method.

Enumeration interface defines the following 2 methods -

```
public boolean hasMoreElements()
public Object nextElement()
```

Limitations of enumeration

- Enumeration concept is applicable only for legacy classes and hence it's not a used with all the collections
- By using enumeration we can get only readAccess & and we can't perform any remove operations.

4.2 Iterator

Iterator concept is applicable **to all the collections.**

While iterating, in addition to read operations we can perform remove operations also.

We can get iterator object by iterator() method of collection interface.

```
Iterator itr = c.iterator()
```

Iterator interface defines following 3 methods

```
public boolean hasNext()
public Object next()
public void remove()
```

Limitations of iterator

- In case of iterator & enumeration we can always move forward direction and we can't move backward direction.
- While performing iterations we can perform only read and remove operations.

- we can't perform replace and add operations.

4.3 ListIterator

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

ListIterator is the child interface of iterator.

While iterating objects by listInterator we can move either to the forward or to the forward or to the backward direction. i.e ListIterator is a bi-directional.

While iterating by listInterator we can perform replace and additions of new objects also.

we can create listInterator objects by using listInterator() of List interface.

ListIterator list = l.listInterator()

This is most powerful cursor and applicable for only Lists.

ListIterator has below 9 methods

public boolean hasNext()
public Object next()
public int nextIndex()
public boolean hasPrevious()
public Object previous()
public int previousIndex()
public void remove()
public void set(Object new)
public void add(Object new)

5 Set

5.1 AbstractSet

5.2 HashSet

5.3 LinkedHashSet

6 SortedSet and NavigableSet

6.1 TreeSet

7 Queue

7.1 AbstractQueue

7.2 PriorityQueue

7.3 BlockingQueue

7.4 PriorityBlockingQueue

7.5 LinkedBlockingQueue

8 Deque

8.1 ArrayDeeue

9 Map

9.1 AbstractMap

9.2 HashMap

9.3 LinkedHashMap

9.4 IdentityHashMap

9.5 WeakHashMap

9.6 Dictionary

9.7 Hashtable

9.8 Properties

10 SortedMap and NavigableMap

10.1 TreeSet

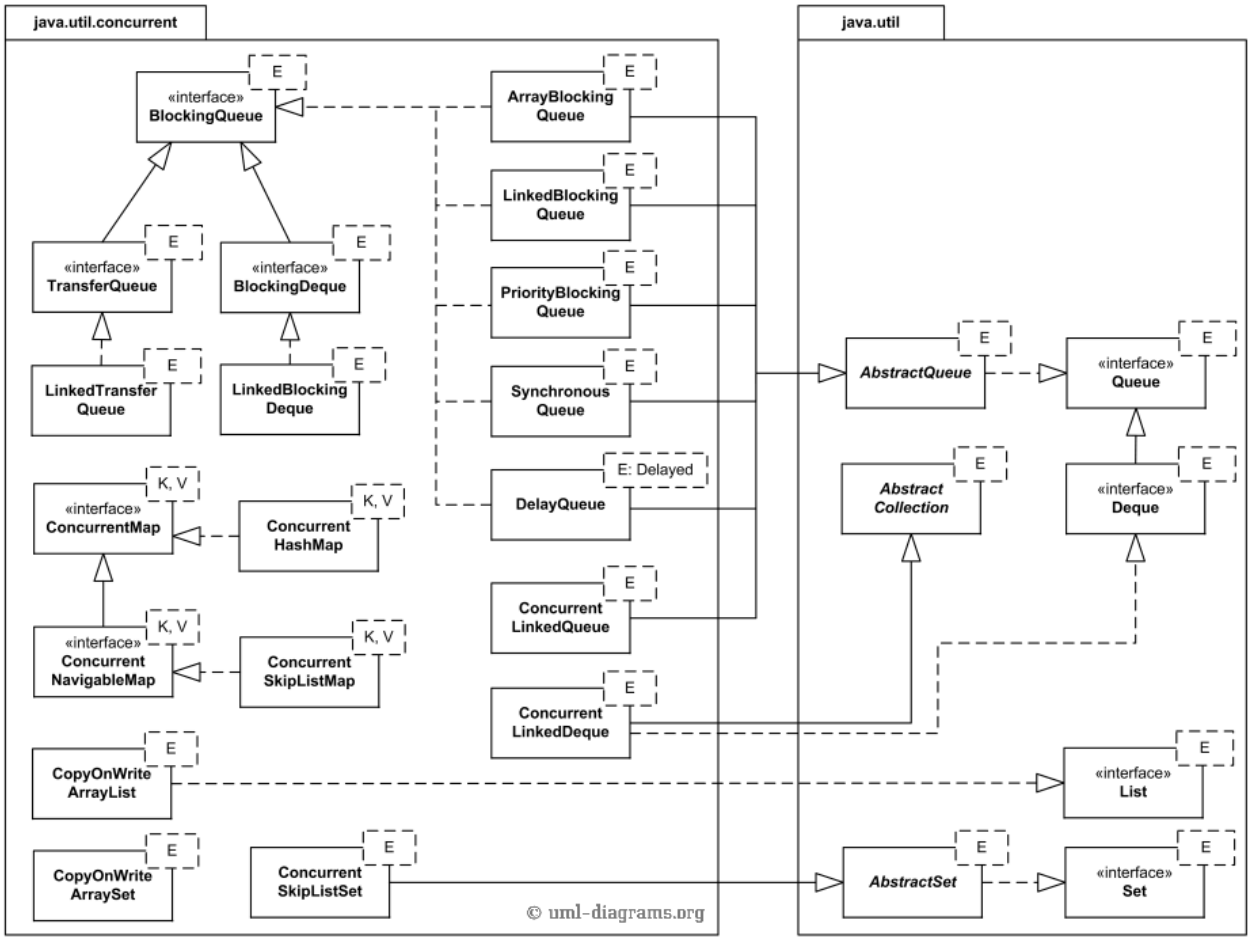
11 Comparable and Comparators

12 Collection Utilities

12.1 Arrays

12.2 Collections

13 Concurrent Collections



13.1 BlockingQueue

13.1.1 ArrayBlockingQueue

13.1.2 DelayQueue

13.1.3 LinkedBlockingQueue

13.1.4 PriorityBlockingQueue

13.1.5 SynchronousQueue

13.1.6 ConcurrentLinkedQueue

13.2 BlockingDeque

13.2.1 LinkedBlockingDeque

13.2.2 ConcurrentLinkedDeque

13.3 ConcurrentMap

13.3.1 ConcurrentHashMap

13.3.2 ConcurrentSkipListMap

13.3.3 ConcurrentNavigableMap

13.4 Concurrent List and Sets

13.4.1 ConcurrentSkipListSet

13.4.2 CopyOnWriteArrayList

13.4.3 CopyOnWriteArraySet