Collections

* Collections that maintain order are called sequencedCollection interface – java21
* Add

Remove

Clear

Contains

isEmpty

iterator

splitIterator

stream

parallelStream

toArray

retain

size- Integer.MAX\_VALUE

* Seq-collections

No DS to collect sequenced data

Fetching of first/last is not maintained uniformly

No proper way of reverse the collection object

Addfirst,addLast,getFirst,getLast,removeFirst,removeLast,putfirst,putLast

* Interface SequencedSet extends SequencedCollection

Interface SequencedSet tries to override the SequencedCollection<E> reversed

method,

SequencedSet is child of SequencedCollection

AL

* List maintains insertion order and supports duplicates
* Interface List extends SequencedCollection

SequencedCollection extends Collection

* AL is unsynchronized
* List<String> list = new ArrayList<>() – diamond is introduced since java7 and follows DRY
* AL – random access is O(1)

Add/remove - needs a shifting operation,

* Collections.unmodifiableList(accountNums) – immutable list

List.of(123,456,789) – won’t allow modification – since Java 9

List.of()

List.of(e1)

List.of(e1,e2)

List.of(elements…) – supports arbitrary/array

* Iterator can remove elements from List in runtime which is an advantage, it supports only reading/removing operations
* ArrayList remove will cause concurrentModification error
* ListIterator can move forward/backward, we can also delete and modify them

ListIterator has performace issues.

* Collections.sort(numbers) – used to sort in ascending order

Collections.sort(numbers,Comparator.reverseOder());

* We can write custom comparators by implementing Comparator interface and overriding compare operation inside it

Collections.*sort*(countries, new LastCharComparator());

*countries.sort(new LastCharComparator());*

* For Comparable,we need to override the compareTo method.

**LinkedList**

* LinkedList is good for frequent insertions/deletions and needs less memory allocation

LinkedList<Integer> intNums = new LinkedList<>();

var countryNames = new LinkedlIst<String>

* Each linkedList has 2 things, one is data and second is reference(pointer) that points to previous and next nodes in list.
* Peek,peekfirst,peekLast, - only for retrieve

Poll,pollFirst,pollLast – both for retrieve and remove

* Linkedlist stores each object in separate link/node, each link/node also stores reference to next link in sequence.In java each LL is also a DLL,as it stores reference to its pre-decessor
* Addition/deletion is fast in LL, Random access reading is expensive in LL as they are not stored sequentially,
* LL is good for reading, DLL is good for addition/deletion of elements

Generics

* Feature that allows to create reusable classes & methods that can work with different datatypes
* True polymorphic code, that works with any data-type
* With Object, we need to do a lot of casting and there is no compile time check
* GenericPair<String,int> - not valid,as primitive cant be used
* Generics can be used for class and interfaces also
* Collections can be used without generics also
* Generics will always use type-safety
* When we write code for parent-supertype, we can use the same for subtype also, this concept is called as Covariance.
* Covar creates lot of runtime issues, hence blocked it for collections
* **Subtype** or upper-bound-wildcards

<? extends T> - a type i.e. either T or child of T, where T is type used as generics of collection

PrintEmployeeNames(ArrayList<? Extends Employee> employees) – at max accepts employee or its sub-class

? indicates some unknown subtye of employee, so we can call it using ArrayList<Employee> or ArrayList<Developer>

* **supertype** or lower-bound-wildcards

contra-variance is opposite of co-variance

we can specify upper-bound for wildcard or lower bound ,but not both

it’s denoted by <?super A>

List<Integer> - accepts only integers, where as

List<?super Integer> - accepts any type that is supertype of Integer, like Number, Object i.e. anything that holds Integer values.

* Unbounded wild-cards <?>

When writing method that relies on object class functionality and

When using methods in generic class indep of type parameter

* Type-safety, no type-casting & code-reusability, compile-time check

Map

* Map extends Generics, but not iterable/collection interface Map<K,>
* Map.Entry is a nested interface within map, that shows single key/value pair
* Get, put and remove are key methods

Map<String,String> = new HashMap<String,String>

HashMap <String,String> = new HashMap<String,String>

var cMap = new HashMap<String,String>();

* LinkedHashMap implements sequenceMap and maintains sequence unlike hashmap
* Map.put(“india”,”delhi”)-

calculate hashCode by invoking hashCode() method of key i.e.

Eg: India.hashcode()->3492

Find bucket-index by using hashcode and length of hashMap i.e., Eg:3

So K,V is placed in bucket number 3

* When 2 or more keys have same hashcode, its called hash collision due to finite hash-codes and infinite keys
* In java8,hashMap was improved to have new DS called TreeNodes, to improve perf incase of collisions and prevent creation of long linkedLists within hashMap buckets
* When no of nodes inside LL exceeds JDK configured number, then LL is converted to TN format
* Static final int TREEIFY\_THRESHOLD = 8;-

Static final int UNTREEIFY\_THRESHOLD = 6;

When LL is having 8 elements, then TREEIFY process will happen and

When TN is only having min of 6 elements, then UNTREEIFY happens

* countryMap.keySet();
* Set<Map.Entry<String,String>> entries = countryMap.entrySet();

Iterator<Map.Entry<String,String>> iterator = entries.iterator();

While(iterator.hasNext()){

Map.Entry<String,String> entry = iterator.next();

String key = entry.getKey();

String value = entry.getValue();

for(Map.Entry<String,String> entry:entries){

String key = entry.getKey();

String value = entry.getValue();

}

* Collection<String> values = countryMap.values();

For(String value:values){

Sout(value);

* TreeMap->impl->NavigableMap->extends->SortedMap->extends->Map
* TM is a red-black tree-based implementation and stores key, value in sorted order based on keys
* TM cant contain null key and maintains insertion order based on key
* TM is efficient in finding first, last keys and finding keys within a specific range
* Public LinkedHashMap<K,V> extends HashMap<K,V> implements SequencedMap<K,V> - combines hash-table and linked-list and maintains order of insertion.
* Immutable maps

Static factory methods

Map.of()

Map.of(k,v)

Map.ofEntries(entry(k1,v1), entry(k2,v2), entry(k3,v3)…)

* In JDK8,Collections.unmodifiableMap(myMap);
* In Java9, Map stringMap = Map.of(“a”,1,”b”,2,”c”,3) -//less than 10 entries

For more than 10 entries , use below one

Map<S,I> myMap = Map.ofEntries(entry(1,”Tom”), entry(2,”Duck”),entry(3,”Harry”));

Set

* Maintains uniqueness, but doesn’t maintain order

Set<String> mySet = new HashSet<>(myList);

HashSet works on hashing mechanism and hash table, it stores only 1 null value, its non-sync and useful for search operation

* Hashset maintans uniqueness by using both hashCode() and equals() method
* Initial capacity – no of buckets, that a hashMap is going to get started-16 by default

If buckets are full, then its going to be incremented by load-factor i.e. 0.75

* Union – hashset1.addAll(hashset2);

Intersection – hashSet1.retainAll(hashSet2);

Difference – hashSet1. removeAll(hashSet2);

Subset – hashSet1.containsAll(hashSet2);

* TreeSet – when we want both unique and sorting in ascending order, - implements Sortedset interface and based on TreeMap and doesn’t allow null values,
* class LinkedHashSet extends HashSet and implements SequencedSet – it maintains insertion order and also implements SequencedSet
* Immutable sets

Static factory methods

Set.of()

Set.of(k,v)

Set.ofEntries(entry(k1,v1), entry(k2,v2), entry(k3,v3)…)

* In JDK8,Collections.unmodifiableSet (myMap);
* In Java9, Map stringMap = Set.of(“a”,1,”b”,2,”c”,3) -//less than 10 entries

For more than 10 entries , use below one

Set<S> myMap = Set.ofEntries(entry(1,”Tom”), entry(2,”Duck”),entry(3,”Harry”));

Enum

* Enum defines organized list of constants in specific order