Java Collections

Java Interview Guide : 200+ Questions

Collection Interface Hierarchy and Methods

- Important Interfaces
  - Collection<E> → Base Interface for all the collections in the hierarchy except Map<K,V>
    - Example interface set<E> extends Collection<E>
  - List<E>, Queue<e>, --> All extend Collection Interface
  - Map<K,V> → Does not extend Collection interface
- Collection
  - Add, remove, size, isEmpty, clear, contains, addAll, removeAll, retainAll, sort
  - Iterator
- What does it mean for a collection to be backed by another.
  - A collection backs another means that changes in one are reflected in the other
    - Ex → LinkHasSet
- List (index is important thing)
  - Operations
    - All methods from Collection
    - Add elements at a specific index Remove element at an index
    - IndexOf LastIndexOf ListIterator
    - Sublist
  - Any implementation of the List Interface would maintain insertion order
    - When A is inserted into a list (without specifying position) and then another elemment is intserted A is stored before B in the list
- Sorting a list
  - Create a class that implement Comparator and implement the public int compare(Cricketer cricketer1) function which return 1,-1,0
  - Collection.sort(<List to be sorted, <new class from previous line>)

Collection Interface and Hierarchy Methods

- Different Type of Lists
  - ArrayList implements RandomAccess (A marker interface meaning it support fast almost constant time access)
    - Slower insertion, deletion
  - Vector → Thread Safe Synchronized Methods and implements RandomAccess. Used when shared between two list
  - LinkedList
    - Elements are doubly linked
    - Iteration is slower then ArrayList
    - Fast Insertion and Deletion
    - Implements the Queue Interface
- Set Operations
  - Does not allow duplication of elements
  - A set that maintains its elements in order --defined by Natural Ordering or through the custom comparator
  - SortedSetInterface → Maintains elements in an sorted order
    - SubSet(fromElement, ToElement) HeadSet(toElement) TailSet(fromElement) First Last
  - NavigableSet → A sorted set that contains operations to find matches
    - Lower , floor, ceiling, higher, pollfirst, pollLast
- Different Types of Sets
  - HashSet unordered and unsorted
    - Iterates in random order by using hashcode
    - Offers constant Time performance for the basic operations assuming the hash function disperses the elements properly among the buckets (Add, contain, remove,size)
    - Focus on capacity (Too High waste both space and time), Too Low (waste time copying the datastructure when has to resize itself)
  - LinkHashSet ordered by order of insertion unsorted and uses hashcode
  - TreeSet implements NavigableSet (Self Ordering Tree) implements the SortedInterface and NavigableSet
    - Access and retrieval are quite fast
    - Implements the SortedSet Interface

Collection Interface Hierarchy and Methods (3)

- Map supports key value pairs
- Operations
  - IsEmpty, containsKey, containsValue, get, remove, putAll, clear, KeySet, EntrySet
- SortedMap – Stores the elements in order of their keys
  - Extends Map<K,V> can supply a Comparator at map creation time.
  - submap(fromKey, toKey), headMap(to Key) and tailMap(fromKey), firstkey, lastKey
- NavigableMap extends SortedMap
  - Reporting closest matches for a leu
  - lowerKey(K key); Map.Entry<K,V> floorEntry) K ceilingKey(K key);0 Map.Entry<K,V> higherEntry(K key), K higherKey(K key);
- HashMap → Implements Map
  - Unsorted, unordered
  - key's hashCode is used
  - Allows a key with a null value
- HashTable → Implements Map
  - Synchronized Thread Safe
  - Unsorted unordered key's hashCode is used does not allow a key with a null value. HashTable does not
- LinkedHashMap insertion order Is maintained (optionally can maintain access order as well)
  - Slow insertion and deletion, but faster insertion
- TreeMap implements Map, NavigableMap – A red black tree with no duplicates
  - Sort order is maintained.

Collection Interface Hierarchy and Methods (4)

- Why Immutable Classes generally the Best candidate for Hashmap Keys
  - Mutable object can have their hashcode change. If the hashcode changes a different bucket might be retrieved.
- Queue Methods
  - add → throws an exception in case of failure
  - offer → returns false for failure
  - remove → throws Exception
  - poll → returns the first element and null if empty
  - element → retrieves but does not remove the head of the queue
  - offer → inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions
- Important interface for Queue
  - Blocking → Supports blocking operations that wait for the queue become non-empty when retrieving an element and wait for space to become available when storing an element
    - Examples : ArrayBlockingQueue, LinkedBlockingQueue
    - Used in Consumer/Producer
      - Allows the consumer to wait
  - Dequeue → Extends the Queue DataStructure to add / remove data at both ends.
  - Priority Queue → ordered according to their natural ordering
- Collection Class
  - binarySearch, reverse, reverseOrder, sort, EmptyList, EmptyMap, EmptySet, CheckedCollection, CheckedType, disjoint
  - emtpyEnumeration, emptyIterator, emptyList, ncopies, replaceAll, reverse, reverseOrder, rotate
  -

What is Generics

- Class, Interfaces and Methods (Parameter and return types) can be Parameterized by types
- Independent of type. Helps to eliminate Casting Exceptions
- Makes type safe code possible
  - If it compiles without any errors or warnings then it must not raise any unexpected ClassCastException during runtime
- Compiling Time Checking
- Provides better reusability
- Example
  - Pubic class LinkList<E> extends AbstractSequentialList<E> implements List<E>, Queue<E>, Clonable, java.io.Seriaizable {
    - Private transient Entry<E>header = new Entry<E>(null,null,null);
    - Private transient int size = 0;
    - Public E getFirst() { if (size == 0) throw NoSuchElemeentException(); return header.next.element;
  - E is the type parameter
  - Developer can do the following LinkList<Integer> li = new LinkList<Integer>();
    - Replace Type Parameter with Concrete Argument (Type Argument)
- What variance is imposed on generic type parameters on generic type parameters. How much control does Java give you over this ?
  - Generic Type Parameters are invariant. For Type A and B G<A> is not a subtype or supertype of G<B>
  - Ex. List<String> is not supertype or subtype List<Object>
    - Will fail to compile
    - List<String> strings = Array.<Object>.asList(“hi there”);
    - List<Object> objects = Arrays.<String>asList(“hi there”);

Subtyping

- Generics and sub-typing
  - ArrayList<Object> ao = new ArrayList<Integer>()
    - Example of why this is not allowed
      - ArrayList<Integer> ai = new ArrayList<Integer>();
        - This is ok
      - ArrayList<Object> ao = ai
      - ao.add(new Object());
        - First element in the array of integers is an object
      - Integer I = ai.get(0)
        - This would result in ClassCastException
      - There is no inheritance relationship between type arguments of a generic class because we cannot guarantee ClassCaseException
      - No inheritance between type arguments
  - The following will work – Inheritance relationship between generic class themselves still exist
    - List<Integer> il2 = new ArrayList<Integer>();
    - Collection<Integer> ci = new ArrayList<Integer>();
    - Collection<String> cs = new Vector<String>(4);
- ArrayList<Number> an = new ArrayList<Number>();
  - an.add(new Integer(5));
  - an.add(new Long(1000L));
  - an.add(new String(“hello”));
    - Compile error
  - Entries in a collection maintain inheritance relationship

Bounded Wildcards

- If you want to bound the unknown type to be a subtype of another type use BoundedWildcard
  - The bounded wildcard can be a Class or Interface
  - Can have multiple constraints : public <T extends Number & Comparable>
  - Example
    - Static void printCollection(Collection<? Extends Number>) c) {
      - For (Object o : c) { System.out.println(o); }
    - Public static void main(String[] args) {
      - Collection<String> cs = new Vector<String>();
      - printCollection(cs) ; //compile error
      - List<Integer> li = new ArrayList<Integer>(10);
      - printCollection(li);
  - Example
    - BWCartridge implements ICartridge
    - ColorCartridge Implements Icartridge
    - Create two Objects Printer<BwCartridge> and Printer<ColorCartridge>
    - printOne(Printer<ICartridge>) will not work
      - The class are not defined with this explicit type
      - Happens because of erasure
    - Reason why extends is needed.
- Use-site Variance → use the ? Extends Type.
  - Public double sum (List<? Extends Number> numbers) {
    - Double sum = 0;
    - For (Number number : numbers) { Sum += number.doubleValue(); } Return sum }
- Even though we define a List of Numbers we can pass in a list of longs – ? supertype allows a method parameter to be contravariant.
  - Allows Callback<Object> to be a subtype of Callback<Number>

Bounded Wildcards

- Upper Bound Wildcard
  - Example
    - public static void process(List<? extends Foo> list) { /* ... */ }
    - The upper bounded wildcard, <? extends Foo>, where Foo is any type, matches Foo and any subtype of Foo. The process method can access the list elements as type Foo:
- Lower Bound Wildcard
  - To write the method that works on lists of Integer and the supertypes of Integer, such as Integer, Number, and Object, you would specify List<? super Integer>.
  -

Generating your own Generic Class

- Even though we define a List of Numbers we can pass in a list of longs – ? supertype allows a method parameter to be contravariant.
- Allows Callback<Object> to be a subtype of Callback<Number>
- Example
  - Public class Pair<F,S> {
    - Public void setFirst(F f) {} }
  - Use: Pair<Number,String> pair<Number,String>();
  - Use: pair.setFirst(5.0);
- Can extend a Generic Class
  - Public class PairExtended<F,S,T> extends Pair<F,S> {
    - T Third
    - Pubic T getThird() { return third; }
- Use a generic class as a type argument
  - ArrayList<Integer> ar4 = new ArrayList<Integer>
  - PairExtended<Number, String, ArrayList<Integer>> pe5 =
    - New PairExtended<Number, String, ArrayList<Integer>>(n4,s4,ar4);
- Generics can also be used in a return type.

Wildcard

- Wildcards Unbounded
  - Use wildcard type argument <?>
  - Collection<?> means collection of unknown type
  - Accessing entries of Collection of unknown type with Object type is safe
  - Example
    - Static void printCollection(Collection<?> c) { for (Object o : c) System.out.println(o); }
    - Public static void main(String[] args) {
      - Collection<String> cs = new Vector<String>();
      - printCollection(cs);
      - List<Integer> li = new ArrayList<Integer>(10);
      - printCollection(li);
  - It is not safe to add arbitrary objects to it however, since we do not know what element type of c stands for we cannot object to it
    - c.add(new Object()); c.add(new String()) would both give use a compile time error
- Difference between a raw type collection and an unbounded wildcard type collection
  - Unbounded means you can use any single type for a data structure (list), but only one type
- 3 types of wild cards
  - Upper Bound – usage of all subtypes as a type parameter – Example extends
  - Lower Bound – usage of all supertypes as type parameter – Example super
  - Unbounded – Enables the usage of all types
    - f(List<T> arg1) { return arg1.get(0) }
    - Can’t have primitive Types
    - Since ? will be considered an object good to have

Raw Type and Type Erasure

- Generic type instantiated with no type arguments
- // Generic type instantiated with type argument
  - List<String> ls = new LinkList<String>();
- // Generic type instantiated with no type argument – This is the raw type
  - List lraw = new LinkedList();
- Type Erasure
  - All generic type information is removed in the resulting byte-code after compilation
  - Generic Type information does not exist during runtime
  - After compilation they all share the same class
    - A class bytecode that represents ArrayList<String> is the same as a class that represents ArrayList
  - Type Safe code → code compiles without warnings and no casts you will never get ClassCastException
- Ways of generating generics
  - Generics are handled during run time and forgotten at compile time. Java
  - Generics are primitives of the system Net
  - Generates real code that operates in that type C++

Interoperability

- Example
  - List<String> ls = new LinkedList<String>
  - ListIraw = ls;
  - lraw.add(new Integer(4)); // Adding an integer to raw time
  - String s = ls.iterator().next(); // Runtime exception will happen here
- Will see the compilation error
  - GenericInteroperability.java uses unchecked or unsafe operations
  - Running the code → ClassCastException

Generics

- Example
  - Class MyListGeneric<T> {
    - private List<T> values;
    - void add(T value) {
      - Values.add(value);
      - }
    - T get(int index) {
      - return values.get(index)
    - }
- Class MyListRestricted<T extends Number>
  - Restricts to a subclass
- Class MyListRestricted<T super Number>
  - Can capture the Class and it parent classes
- Example of a Generic Method
  - Static < X extends Number> X doSomething(X number) {
    - X result = number;
    - Return result;
  - }

Concurrent Collections

- Difference between synchronized and concurrent collections
  - Synchronized Collection
    - Use synchronized methods and blocks
    - Only one thread executing at any time
    - Examples HashTable and Vector
  - Unsynchronized Collections
    - Synchronizing on some object that naturally encapsulates the set
    - Collection.synchronizedSet or synchronizedSortedSet and Collections.synchronizedList(new ArrayList() or new LinkList())
  - Concurrent Collections
    - New approaches to synchronization
      - Copy on Write
      - Compare and Swap
      - Locks
    - Copy on Write
      - All values in collection are stored in an internal immutable (non-changeable) array.
        - A new array is created if there are any modification to the collection
      - Read operations are not synchronized. Only write operations are synchronized
      - Used where reads greatly out number write's on a collection
        - CopyOnWriteArrayList and CopyOnWriteArraySet are implementations of this approach
      - Observer scenarios are example
    - Compare and Swap
    - ` Compares the memory location original value with a the value retrieved before the operation
      - If they are the same then it changes else it does not – Atomic Operation
      - Instead of synchronizing entire method the value of the member variable before calculation is cached
      - Used by ConcurrentLinkedQueue
      - After calculation the cached value is compared with the current value
        - Same → Keep old value return false
        - Different →Swap Value return true

Iterators
Why Collection don't extend Cloneable and Serializable Interfaces
Natural Ordering

- Unsupported Operation → When you try to modify an immutable list
- Fail-Safe vs Fail Fast Iterators
  - Fail Fast iterators throw a ConcurrentModificationException if there is a modification to the underlying collection is modified
    - Example ArrayList, HashSet, HashMap. In other most collections.
  - Fail Safe Iterators – Do not throw iterators do not throw exception, but Takes a copy of the data structure and iterates over that
    - All collection class in java.util are fail-fast while java.util.concurrent are fail safe
    - Algorithm → Collection has a mods property which is checked whenever the next value is retrieved
    - Single Thread → Structure is modified at any time by any method other than the iterator's own remove method
    - Multiple Threaded Environment → If one thread is modifying the structure of the collection while other thread is iterating over it.
    - Example ConncurrentHashMap, CopyOnWriteArrayList
- ListIterator → Extends Iterator and allows bidirectional transversal
- Why Collection don't extend Cloneable and Serializable Interface
  - Some allow duplicate keys while other don't
  - The concrete implementations should decide how they can be cloned or serialized.
  - Example : what does it mean to clone a collection that is backed by a terabyte SQL
- Nature Ordering
  - Has natural ordering if it has implement java.lang.Comparable
  - The CompareTo is referred to as its natural comparison method
  - Class with natural Ordering
    - String Alphabetical
    - Date Chronological Order
    - Integer Numerical Order

Utility Collection Classes

- Provides methods to return an empty collection
  - Collection.emptyList(), Collection.empetySet(), Collection.emptyMap()
- Provides methods for sorting
  - Reorders a List so that elements are in ascending order according to an ordering relationship
  - Example List l = Collections.sort(l);
  - Using Natural Ordering
    - Inputs
      - Sort a list based on natural ordering
      - Sort a list based on the Comparator provided
      - Sort a list on elements that do not implement comparable then a ClassCast Exception will be thrown.
    - Using Comparator
      - Comparators can be passed to a sort method to allow precise control over the sort order
      - Can be use dto control the order of certain data structures (sorted sets)
      - Provide order for collection of objects that do not have anatural ordering.
- Provides methods for shuffling
  - Destroy any trace of order

Utility Collection Classes (2)

- Routine Data Manipulation
  - Reverses the array
  - Rewrites every element in the list with the specified value
  - Overwrites the destination with the sort list
  - Swap the elements in the list
  - Add elements individually or by array
  - Frequency → How many times the same element aapears
  - Disjoint → returns true if there are no elements in common
- Searching
  - Call binarySearch()
- Composition
- Find Extreme Values

Synchronized Collections

- Synchronized Already HashTable and Vector
  - PutIfAbsent
  - RemoveIfPresent
- Regular ArrayList, HashMap
- Concurrent Concurrent[Hashmap,ArrayList,ArraySet]
  - Concurrent Iterate and Modify across current threads
  - Hashmap
    - Lock Threading : When one thread iterates and another modifies they will sync up at one point.
  - ArrayList, ArraySet
    - Clone every time a modification is made.
  - Cannot synchronized
  - Should do more reads than writes
- Concurrent Modification Exception
  - When two thread try to access a regular collection.
  - If 9 threads are reading and one write you get this exception
- Different between concurrent and synchronized collections
  - Both used for thread safety
  - Performance – the concurrent collection is better than that of synchronized collection
  - Synchronized – acquires a lock on the whole collection
  - Concurrent acquires a lock on a segment of the collection which the other segments remain open for reads and writes
  - When to use
  - Synchronized – Iterate through the whole collection and be sure that nothing changes while iterating
  - Concurrent used when multiple threads are reading and writing to a collection and want to preform read and writes as fast as we can..