**Java Collection - 2022**

**Java Collection Highlights**

|  |  |  |
| --- | --- | --- |
| **Type** | **Normal** | **Concurrent** |
| List | ArrayList | [**CopyOnWriteArrayList**](file:///E:\dev\java7\docs\api\java\util\concurrent\CopyOnWriteArrayList.html) |
|  | [LinkedList](file:///E:\dev\java7\docs\api\java\util\LinkedList.html) | [**CopyOnWriteArrayList**](file:///E:\dev\java7\docs\api\java\util\concurrent\CopyOnWriteArrayList.html) |
| Set | HashSet | [**CopyOnWriteArraySet**](file:///E:\dev\java7\docs\api\java\util\concurrent\CopyOnWriteArraySet.html) |
|  | TreeSet | [**ConcurrentSkipListSet**](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListSet.html) |
| Map | HashMap | [**ConcurrentHashMap**](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentHashMap.html) |
|  | TreeMap | [**ConcurrentSkipListMap**](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentSkipListMap.html) |
| Queue | LinkedList | [**ConcurrentLinkedQueue**](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentLinkedQueue.html)**,**  [**ArrayBlockingQueue**](file:///E:\dev\java7\docs\api\java\util\concurrent\ArrayBlockingQueue.html) |
|  | [PriorityQueue](file:///E:\dev\java7\docs\api\java\util\PriorityQueue.html) | [**PriorityBlockingQueue**](file:///E:\dev\java7\docs\api\java\util\concurrent\PriorityBlockingQueue.html) |
| Deque | [ArrayDeque](file:///E:\dev\java7\docs\api\java\util\ArrayDeque.html) | [LinkedBlockingDeque](file:///E:\dev\java7\docs\api\java\util\concurrent\LinkedBlockingDeque.html),  [ConcurrentLinkedDeque](file:///E:\dev\java7\docs\api\java\util\concurrent\ConcurrentLinkedDeque.html), |

**ArrayList**

Initial capacity of an ArrayList is 10. The capacity is calculated like the following formula.

**int newCapacity = (oldCapacity \* 3)/2 + 1 🡺 Deprecated**

**int newCapacity = ArraysSupport.newLength(oldCapacity, minCapacity - oldCapacity, oldCapacity >> 1;**

**int** prefLength = oldLength + Math.*max*(minGrowth, prefGrowth);

**int** oldCapacity = elementData.length;

**int oldCapacity = elementData.length; 🡺 use this for answering in interview**

**int newCapacity = oldCapacity + (oldCapacity >> 1);**

**Important methods of ArrayList**

**retainAll(): This method is used to contain the intersection of the two list.**

**List<String> list1 = new ArrayList<String>(Arrays.asList("A", "B", "C"));**

**List<String> list2 = new ArrayList<String>(Arrays.asList("B", "C", "D", "E", "F"));**

**list1.retainAll(list2); 🡺 list1 Contains intersection of elements**

System.*out*.println(list1); // [B, C]

System.*out*.println(list2); // [B, C, D, E, F]

list2.retainAll(list1); **🡺 list2 Contains intersection of elements**

System.*out*.println(list1); // [A, B, C]

System.*out*.println(list2); // [B, C]

**What is the difference between Queue.poll() and Stack.pop() ?**

**Stack.pop() 🡺 Provides Last element.** Use **Stack** to get things out in the **reverse order**.

**Queue.poll() 🡺 Provides First element.** Use **Queue** to get things out in the **insertion order**.

LinkedList<Integer> ll = new LinkedList<>();  
ll.add(1); ll.add(2); ll.add(3); ll.add(4); ll.add(5);

System.*out*.println("Peek: "+ ll.peek()); *// 1*System.*out*.println("Poll: "+ll.poll()); *// 1*System.*out*.println(ll); *// 2, 3, 4, 5*System.*out*.println("Pop: "+ ll.pop()); *// 2*System.*out*.println(ll); *// 3,4,5*

**Stack (Legacy)** Important Methods for **Stack** 🡺 **push()** and **pop() 🡺 Stack is LIFO**

The Stack class represents a **last-in-first-out (LIFO)** stack of objects.

**Examples** 🡺 Achieving the undo operation in notepads, Browser back button uses a Stack.

Stack<String> stack = **new** Stack<>();  
stack.push("A1"); 🡸 First Element  
stack.push("B2");  
stack.push("C3");  
stack.push("D4");  
stack.push("E5"); 🡸 Last Element  
System.***out***.println("Simply Stack: "+stack); // [A1, B2, C3, D4, E5]  
stack.forEach(a-> System.***out***.print(a+",")); //A1,B2,C3,D4,E5  
System.***out***.println("**\n**Value poped from stack");  
**while**(!**stack.isEmpty()**) {  
 System.***out***.print(**stack.pop()**+"**\t**"); // E5 D4 C3 B2 A1 🡺 Pops Last element  
}  
System.***out***.println("**\n**Now Stack length: "+stack.size()); // Now Stack length: 0

The output of Stack will be same if you use push() or add() method. The data will be stored in sequential manner.

**Only retrieval is important, in case of pop(), it retrieves the last element only.**

In case of Stack, it has methods like add(), push(), pop() and get()

**Queue** Important Methods for **Queue** 🡺 **offer()** and **poll() 🡺** Queue is **FIFO**.

**Examples** 🡺 Print of papers in Printer, Call Centers handling the call.

Queue<String> queue = **new** LinkedList<>();  
queue.offer("A1");  
queue.offer("B2");  
queue.offer("C3");  
queue.offer("D4");  
queue.offer("E5");  
System.***out***.println("Simply Queue: "+queue); // [A1, B2, C3, D4, E5]  
queue.forEach(a-> System.***out***.print(a+",")); // A1,B2,C3,D4,E5,  
System.***out***.println("**\n**Value polled from Queue");  
**while**(!**queue.isEmpty()**) {  
 System.***out***.print(**queue.poll()**+"**\t**"); // A1 B2 C3 D4 E5 🡺 Polls First Element  
}  
System.***out***.println("**\n**Now Queue length: "+queue.size()); // Now Queue length: 0

**It does not matter whether you use add() or offer() method in Queue, the data will be stored in sequential manner.** **Only retrieval is important**. You can use LinkedList or PriorityQueue for a queue implementation. **poll() method always returns the first element in the structure**. Poll() method retrieves the value and then deletes the value.

Example on PriorityQueue.

PriorityQueue<Integer> pq = **new** PriorityQueue<>(); // Comparator.naturalOrder()  
pq.add(1);pq.add(2);pq.add(3);pq.add(4);pq.add(5);  
System.***out***.println(pq); // 1, 2, 3, 4, 5  
System.***out***.println("Value from priority queue");  
**while**(!pq.isEmpty()) {  
 System.***out***.print(pq.poll()+"**\t**"); // 1 2 3 4 5   
}

If you use like this

PriorityQueue<Integer> pq = **new** PriorityQueue<>(Comparator.*reverseOrder*());

**while**(!pq.isEmpty()) {  
 System.***out***.print(pq.poll()+"**\t**"); // 5 4 3 2 1   
}

Consider a real-life scenario: In a hospital, patients are attended to based on the severity of their conditions, not just their arrival time. This is a classic real-world demonstration of a Priority Queue.

How to set Priority

**public record** Patient(String name, **int** serverity) {  
}

**Based upon Low Severity Case of Patient**

**Comparator<Patient> byLowSeverity = Comparator.*comparing*(p->p.serverity());  
PriorityQueue<Patient> pq = new PriorityQueue<>(byLowSeverity);**  
pq.add(**new** Patient("John", 7));  
pq.add(**new** Patient("Shyam", 5));  
pq.add(**new** Patient("Pili", 1));  
pq.add(**new** Patient("Kiran", 7));  
System.***out***.println(pq); // [Patient[name=Pili, serverity=1], Patient[name=John, serverity=7], Patient[name=Shyam, serverity=5], Patient[name=Kiran, serverity=7]]  
  
System.***out***.println("Based upon Low Severity");  
**while**(!pq.isEmpty()) {  
 System.***out***.println(pq.poll());  
}

OUTPUT

Patient[name=Pili, serverity=1]

Patient[name=Shyam, serverity=5]

Patient[name=Kiran, serverity=7]

Patient[name=John, serverity=7]

**Based upon High Severity Case of Patient 🡺 Also for highest rated items**

**Comparator<Patient> byHighSeverity = Comparator.*comparing*(p->p.serverity(), Comparator.*reverseOrder*());  
PriorityQueue<Patient> pq = new PriorityQueue<>(byHighSeverity);**pq.add(**new** Patient("John", 7));  
pq.add(**new** Patient("Shyam", 5));  
pq.add(**new** Patient("Pili", 1));  
pq.add(**new** Patient("Kiran", 7));  
System.***out***.println(pq); // [Patient[name=John, serverity=7], Patient[name=Kiran, serverity=7], Patient[name=Pili, serverity=1], Patient[name=Shyam, serverity=5]]  
  
System.***out***.println("Based upon High Severity");  
**while**(!pq.isEmpty()) {  
 System.***out***.println(pq.poll());  
}

OUTPUT

Patient[name=John, serverity=7]

Patient[name=Kiran, serverity=7]

Patient[name=Shyam, serverity=5]

Patient[name=Pili, serverity=1]

From the above, as you can see only retrieval method is important ie poll(). It does not matter it displays the data.

**Deque**

It is a linear collection that supports **element insertion and removal at both ends**. ArrayDeque() : Constructs an empty array deque with an initial capacity sufficient to hold 16 elements.

Examples 🡺 **Undo/Redo functionality**, **Browser history**, **Palindrome checker**

**If ArrayDeque used as Stack**

ArrayDeque<Integer> deque = new ArrayDeque<>();

deque.push(1);  
deque.push(2);  
deque.push(3);  
deque.push(4);  
deque.push(5);

System.*out*.println(deque); // 🡺 **[5, 4, 3, 2, 1]**  
**while(!deque.isEmpty()) {  
 System.*out*.print(deque.pop()+"\t"); // 5 4 3 2 1  
}**

**In case of ArrayDeque as Stack, the data are organized in the reverse order of insertion**.

If you need high performance for enqueue and dequeue operations and do not require null elements, ArrayDeque is typically the better choice.

**If ArrayDeque used as Queue**

ArrayDeque<Integer> deque = new ArrayDeque<>();

deque.offer(1);  
deque.offer(2);  
deque.offer(3);  
deque.offer(4);  
deque.offer(5);

System.***out***.println(deque); // 1 2 3 4 5  
**while**(!deque.isEmpty()) {  
 System.***out***.print(deque.poll()+"**\t**"); // 1 2 3 4 5  
}

**In case of ArrayDeque as Queue, the data are organized in the order of insertion**.

**What is difference between LinkedList and PriorityQueue ?**

* The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
* LinkedList preserves the insertion order, PriorityQueue does not

**HashSet:** Constructs a new, empty set; the backing HashMap instance has default **initial capacity (16) and load factor (0.75**).

**Explain the differences among Enumeration, Iterator and ListIterator Enumeration**

It is a legacy interface which generates a series of elements, one at a time. Successive calls to the nextElement method return successive elements of the series. To print all elements of a Vector<E> v:

**for (Enumeration<E> e = v.elements(); e.hasMoreElements();)**

**System.out.println(e.nextElement());**

It has only two methods, **hasMoreElements()** and **nextElement()**

**ListIterator**

It allows you to traverse in both the direction. The functionality of Enumeration interface is duplicated by the Iterator interface. **Iterator has a remove()** method while Enumeration doesn't. Example given below.

List<Integer> list = List.*of*(1,2,3,4,5);  
ListIterator li = list.listIterator();  
**while(li.hasNext()) {  
 System.*out*.println("Next value: "+li.next()); 🡺 1,2,3,4,5  
}**  
**while(li.hasPrevious()) {  
 System.*out*.println("Previous Value: "+li.previous()); 🡺 5, 4, 3,2,1  
}**

**What is the difference between Collection and Collections ?**

**Collection is an interface . Collections is a class which provides utility methods**.

Some of the significant methods are given below.

**max(Collection<? extends T> coll)** : Returns the maximum element of the given collection,

according to the natural ordering of its elements.

**max(Collection<? extends T> coll, Comparator<? super T> comp)** : Returns the maximum element of the given collection, according to the order induced by the specified comparator.

Similarly **min()** method has been given. These methods are useful when you have employee objects and you want to calculate the min and max age.

**shuffle(List<?> list)** : Randomly permutes the specified list using a default source of randomness.

**sort(List<T> list)** : Sorts the specified list into ascending order, according to the natural ordering of its elements. sort(List<T> list, Comparator<? super T> c) : Sorts the specified list according to the order induced by the specified comparator.

**What is the difference between Comparable and Comparator and when to use in situations ?**

**Comparable** is used for natural order sorting or called the default sorting. Comparator is used for custom order sorting.

Comparable interface has a method called "**compareTo(Object)**" (ie comprable to ), it has the following structure.

**public class** Employee **implements Comparable<Employee>** {  
  
 **private** String name;  
 **private int** age;  
 **private double** sal;  
  
 **// Constructor**  
  
 @Override  
 **public int** compareTo(Employee emp) {  
// return Integer.compare(this.age, emp.age);  
// return Double.compare(this.sal,emp.sal); // In Ascending  
// return Double.compare(emp.sal, this.sal); // In Descending  
// return this.name.compareTo(emp.name); // In Ascending  
 **return** emp.name.compareTo(**this**.name); // In Descending  
 }  
  
}

**Comparator** interface has a method called "**compare(Object,Object)**" and it has the following structure.

**Ascending order**

**Comparator<Employee> ageComparator = (a,b) -> a.getAge() > b.getAge() ? 1 : -1;**

**Descending Order**

**Comparator<Employee> ageComparator = (a,b) -> a.getAge() < b.getAge() ? 1 : -1;**

Both the interface will be used in the specific situations like , we have a list of employee objects and the finance department of an organization will sort the list of employee objects based upon the salary for the income tax calculation where as other departments like HR and Personal departments will sort according to the age of the employees or based upon the employee seniority. In this type of cases, the default sorting will be based on salary for which we have to implement comparable interface. For any other custom sorting we have to use Compartor interface.

**How to use Comparator in Java 8**

*//Ascending***Comparator<Employee> ageComparator = (a,b) -> a.getAge() > b.getAge() ? 1 : -1;**  
*//Descending***Comparator<Employee> ageComparator = (a,b) -> a.getAge() < b.getAge() ? 1 : -1;**  
Collections.*sort*(empList, ageComparator);

**Significance of toString() method**

The toString method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character '@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

**getClass().getName() + '@' + Integer.toHexString(hashCode())**

**Fail Fast Vs Fail Safe Iterators in Java**

**Fail-Fast**

"Fail fast" means: it may fail ... and the failure condition is checked aggressively so that the failure condition is detected before damage can be done. **The fail-fast iterators are typically implemented using a counter (or something) on the list object**. When the list is updated, the counter is incremented. When an Iterator is created, the current value of the counter is embedded in the Iterator object. When an Iterator operation is performed, the method compares the two counter values and throws a CME if they are different. Iterator in **ArrayList are Fail-Fast**.

**Fail Safe**

The only difference is fail-safe iterator doesn't throw any Exception, contrary to fail-fast Iterator, if Collection is modified structurally while one thread is iterating over it. This is because they work on clone of Collection instead of original collection and that’s why they are called as **fail-safe iterator**. **Iterator of CopyOnWriteArrayList is an example of fail-safe Iterator also iterator written by ConcurrentHashMap keySet is also fail-safe iterator and never throw ConcurrentModificationException in Java**.

But in this approach we can find the following two issues:

a. The overhead of copying the data structure (time and memory)

b. The fail-safe iterator doesnt guarantees the data being read is the data currently in the data structure with the created collection.

**Array vs linked list in Java**

* LinkedList does not support random access while Array supports
* Array needs contiguous memory allocation, which may result in java.lang.OutOfMemoryError.
* Linked list is distributed data structure, it's element are scattered over heap and doesn't need a contiguous memory allocation.
* Array is a fixed length data structure.
* Inserting element at beginning of linked list, and deleting element from end of linked list is O(1) operation
* Array is ideal for implementing fast caches e.g. HashMap or Hashtable, which requires constant time retrieval.
* Array can be one or multi-dimensional, while linked list can be singly, doubly or circular linked list

**LinkedList vs ArrayList in Java and When to use which one**

* **ArrayList provides constant time for search operation, so it is better to use ArrayList if searching is more frequent operation than add and remove operation**.
* The LinkedList provides constant time for add and remove operations. So it is better to use LinkedList for manipulation.
* **For frequent insertion and deletion, LinkedList is good** and there is no resizing of array.
* LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.

**LinkedList**

* **get(int index) is O(n)** (with n/4 steps on average), but O(1) when index = 0
* **add(int index, E element) is O(n)** (with n/4 steps on average), but O(1) when index = 0
* **remove(int index) is O(n)** (with n/4 steps on average), but O(1) when index = 0
* **Iterator.remove() is *O(1)***. **One of the main benefits of** LinkedList<E>
* **ListIterator.add(E element) is *O(1)***. **One of the main benefits of** LinkedList<E>
* **LinkedList<E> allows for constant-time insertions or removals using iterators**

**ArrayList**

* **get(int index) is *O(1)***. **Main benefit of** ArrayList<E>
* **add(E element) is *O(1)***amortized, but *O(n)* worst-case since the array must be resized and copied
* add(int index, E element) is *O(n)* (with *n/2* steps on average)
* remove(int index) is *O(n)* (with *n/2* steps on average)
* Iterator.remove() is *O(n)* (with *n/2* steps on average)
* ListIterator.add(E element) is *O(n)* (with *n/2* steps on average)

**Amortized meaning: Gradual increase.**

**How RandomAccess interface is used as MarkerInterface?**

**Refer to the class AbstractList.java**

**public** List<E> subList(**int** fromIndex, **int** toIndex) {  
 **return** (**this instanceof** RandomAccess ?  
 **new** RandomAccessSubList<>(**this**, fromIndex, toIndex) :  
 **new** SubList<>(**this**, fromIndex, toIndex));

**What is Red-Black Tree Algorithm: 🡺 TreeMap uses Red-Black Tree Algorithm 🡸**

In computer science, a red–black tree is a kind of self-balancing binary search tree. **Each node stores an extra bit representing "color" ("red" or "black"), used to ensure that the tree remains balanced during insertions and deletions**.

**Rules That Every Red-Black Tree Follows:**

1. **Red/Black Property:** Every node is colored, either red or black.
2. **Root Property:** The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.
4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

Search 🡺 O(log n) ,

Insert 🡺 O(log n) 🡺 **n** is the total number of elements in the red-black tree

Delete 🡺 O(log n)

