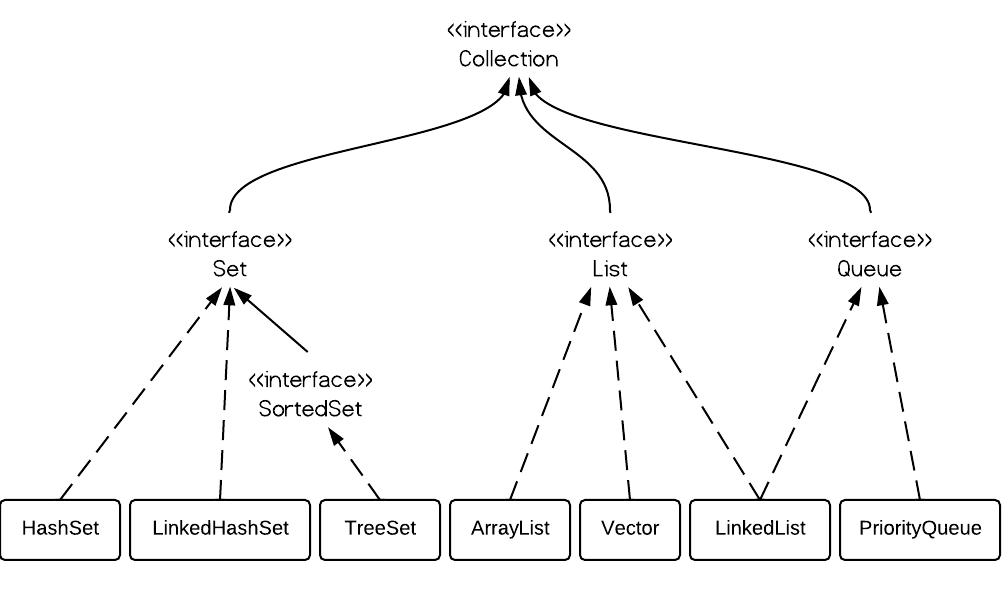
JDK API Question

1. LinkedList vs ArrayList



\* ArrayList: Basically array, dynamic increasing 50%, not thread safe. Get/Set fast because it is array.

\* LinkedList: Doubly linkedlist implemented, so it has poll(), peek(), offer(); Remove/Add fast. Get/Set slow.

\* Vector: array, dynamic increasing 100% size, thread safe.

Vector vs ArrayList: Vector is slow and take more space. Most of time, developer use locks to lock a few instruction, so does not need thread-safe data structure like vector.

2.Create two dimensional List, which syntax is correct?

The following are ok.

List<List<Integer>> table4 = new ArrayList<List<Integer>>();

List<List<Integer>> table1 = new ArrayList<List<Integer>>();

List<ArrayList<Integer>> table2 = new ArrayList<ArrayList<Integer>>();

ArrayList<ArrayList<Integer>> table3 = new ArrayList<ArrayList<Integer>>();

The following is wrong,

List<**List**<Integer>> table = new ArrayList<ArrayList<Integer>>();

Rule of thumb, the inner type must be identical from left to right. The outter type in the rhs needs to be solid type not interface.

3. Iterate map

There are a few ways to do this,

\* EntrySet and loop,

public void iterateUsingEntrySet(Map<String, Integer> map) {

    for (Map.Entry<String, Integer> entry : map.entrySet()) {

        System.out.println(entry.getKey() + ":" + entry.getValue());

    }

}

\* Using keySet()

for (Integer key : customers.keySet()) {

System.out.println("Key : " + key + " value : " + customers.get(key));

}

\* EntrySet and Iterator

public void iterateUsingIteratorAndEntry(Map<String, Integer> map) {

    Iterator<Map.Entry<String, Integer>> iterator = map.entrySet().iterator();

    while (iterator.hasNext()) {

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

        System.out.println(entry.getKey() + ":" + entry.getValue());

    }

}

\* KeySet and Iterator

Iterator<Integer> iterator = customers.keySet().iterator();

while (iterator.hasNext()) {

Integer key = iterator.next();

System.out.println("Key : " + key + " value : " + customers.get(key));

}

\* With Lambda(forEach)

public void iterateUsingLambda(Map<String, Integer> map) {

    map.forEach((k, v) -> System.out.println((k + ":" + v)));

}

\* Stream API

public void iterateUsingStreamAPI(Map<String, Integer> map) {

    map.entrySet().stream()

      // ...

      .forEach(e -> System.out.println(e.getKey() + ":" + e.getValue()));

}

Another example,

map.entrySet().stream()

.filter(x -> "Jan".equals(x.getValue()))

.forEach( x -> System.out.println("Key : " + x.getKey() + " Value : " + x.getValue()));

1/ Sort Map by Key

Use treemap, first build a hashmap, and then add hashmap object as parameter to treemap constructor.

Map<String, String> unsortMap = new HashMap<String, String>();

Map<String, String> treeMap = new TreeMap<String, String>(unsortMap);

Then when you iterator each entry, their key is sorted.

2/ Sort Map by Value

Still uses TreeMap, but you have to implement a comparator object.

Map<Integer, String> treeMap = new TreeMap<Integer, String>(

new Comparator<Integer>() {

@Override

public int compare(Integer o1, Integer o2) {

return o2.compareTo(o1);

}

});

/\* For Java 8, try this lambda

Map<Integer, String> treeMap = new TreeMap<>(

(Comparator<Integer>) (o1, o2) -> o2.compareTo(o1)

);

\*/

treeMap.putAll(unsortMap);

printMap(treeMap);

4. Collections.sort() usage.

Sort a list of object by its property value.

**public** **static** **void** main(String[] args) **throws** Exception {

List<Movie> m = **new** ArrayList<>();

m.add(**new** Movie("a", 3));

m.add(**new** Movie("b", 5));

m.add(**new** Movie("c", 4));

m.add(**new** Movie("d", 1));

Collections.*sort*(m, **new** Comparator<Movie>() {

**public** **int** compare(Movie m1, Movie m2) {

**return** (**int**) (m1.rate - m2.rate);

}

});

m.forEach(e-> System.***out***.println(e.rate));

}

5.　java.util.Arrays.sort(...)

public static void sort(int[] arr, int from\_Index, int to\_Index)

eg:

**int**[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

Arrays.sort(arr);

6. java.util.Arrays.asList

public static <T> List<T> asList(T... a)

eg:

String a[] = new String[]{"abc","klm","xyz","pqr"};

List list1 = Arrays.asList(a);

7. ArrayDeque vs Linkedlist

An ArrayDeque (also known as an “Array Double Ended Queue”, pronounced as “ArrayDeck”) . If you need to use queue or stack in your programming for work or coding for interview, this is a very important and useful data structure to use.

When you use it to implement queue, the first is on the left and last is on the right. (FIFO). First is head and last is tail.

When you use it to implement as stack, then you need to add to the last and remove from the last.

It has an abundant of APIs as the following,

addFirst/getFirst/removeFirst,

addLast/getLast/removeLast,

all these methods throws exceptions, and the followings are not throwing exceptions.

[offer/peek/poll]First/Last()

There are also push()/pop()/offer()/poll()

poll() removes the head.

### **Using ArrayDeque as a Stack**

@Test

**public** **void** whenPush\_addsAtFirst() {

    Deque<String> stack = **new** ArrayDeque<>();

    stack.push(**"first"**);

    stack.push(**"second"**);

    assertEquals(**"second"**, stack.getFirst());

}

@Test

**public** **void** whenPop\_removesLast() {

    Deque<String> stack = **new** ArrayDeque<>();

    stack.push(**"first"**);

    stack.push(**"second"**);

    assertEquals(**"second"**, stack.pop());

}

### **Using ArrayDeque as a Queue**

@Test

**public** **void** whenOffer\_addsAtLast() {

    Deque<String> queue = **new** ArrayDeque<>();

    queue.offer(**"first"**);

    queue.offer(**"second"**);

    assertEquals(**"second"**, queue.getLast());

}

@Test

**public** **void** whenPoll\_removesFirst() {

    Deque<String> queue = **new** ArrayDeque<>();

    queue.offer(**"first"**);

    queue.offer(**"second"**);

    assertEquals(**"first"**, queue.poll());

}

8. The usage of priority queue.

9. HashMap, LinkedHashMap and HashMap’s implementation

LinkedHashMap

Java LinkedHashMap class is Hash table and Linked list implementation of the Map interface, with predictable iteration order. It inherits HashMap class and implements the Map interface.

The important points about Java LinkedHashMap class are:

* A LinkedHashMap contains values based on the key.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is same as HashMap instead maintains insertion order.

**public** **class** LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>

**public** **static** **void** main(String args[]) {

LinkedHashMap<Integer, String> hm = **new** LinkedHashMap<Integer, String>();

hm.put(100, "Amit");

hm.put(100, "Amitxxxx");

hm.put(101, "Vijay");

hm.put(102, "Rahul");

hm.put(**null**, "Rahul");

hm.put(**null**, "Rahulxxx");

**for** (Map.Entry m : hm.entrySet()) {

System.***out***.println(m.getKey() + " " + m.getValue());

}

System.***out***.println(hm.size());

}

4 elements including null key element.

Hashmap in the following will behave the same except not in order,

**public** **static** **void** main(String args[]) {

HashMap<Integer, String> hm = **new** HashMap<Integer, String>();

hm.put(100, "Amit");

hm.put(100, "Amitxxxx");

hm.put(101, "Vijay");

hm.put(102, "Rahul");

hm.put(**null**, "Rahul");

hm.put(**null**, "Rahulxxx");

**for** (Entry<Integer, String> m : hm.entrySet()) {

System.***out***.println(m.getKey() + " " + m.getValue());

}

System.***out***.println(hm.size());

}

Output:

null Rahulxxx

100 Amitxxxx

101 Vijay

102 Rahul

4

10. Autoclosable and try-with-resource

Support for try-with-resources – introduced in Java 7 – allows us to declare resources to be used in a try block with the assurance that the resources will be closed when after execution of that block. The resources declared must implement the AutoCloseable interface.

**try** (PrintWriter writer = **new** PrintWriter(**new** File(**"test.txt"**))) {

    writer.println(**"Hello World"**);

}

Originally you will write code like this,

Scanner scanner = **null**;

**try** {

    scanner = **new** Scanner(**new** File(**"test.txt"**));

**while** (scanner.hasNext()) {

        System.out.println(scanner.nextLine());

    }

} **catch** (FileNotFoundException e) {

    e.printStackTrace();

} **finally** {

**if** (scanner != **null**) {

        scanner.close();

    }

}

Now with try-with-resource, you write it like,

**try** (Scanner scanner = **new** Scanner(**new** File(**"test.txt"**))) {

**while** (scanner.hasNext()) {

        System.out.println(scanner.nextLine());

    }

} **catch** (FileNotFoundException fnfe) {

    fnfe.printStackTrace();

}

**With multiple resources,**

**try** (Scanner scanner = **new** Scanner(**new** File(**"testRead.txt"**));

    PrintWriter writer = **new** PrintWriter(**new** File(**"testWrite.txt"**))) {

**while** (scanner.hasNext()) {

    writer.print(scanner.nextLine());

    }

}

## A Custom Resource with AutoCloseable

**public** **class** MyResource **implements** AutoCloseable {

    @Override

**public** **void** close() **throws** Exception {

        System.out.println(**"Closed MyResource"**);

    }

}

## **Resource Closing Order**

Resource1

**public** **class** AutoCloseableResourcesFirst **implements** AutoCloseable {

**public** AutoCloseableResourcesFirst() {

        System.out.println(**"Constructor -> AutoCloseableResources\_First"**);

    }

**public** **void** doSomething() {

        System.out.println(**"Something -> AutoCloseableResources\_First"**);

    }

    @Override

**public** **void** close() **throws** Exception {

        System.out.println(**"Closed AutoCloseableResources\_First"**);

    }

}

resource2

**public** **class** AutoCloseableResourcesSecond **implements** AutoCloseable {

**public** AutoCloseableResourcesSecond() {

        System.out.println(**"Constructor -> AutoCloseableResources\_Second"**);

    }

**public** **void** doSomething() {

        System.out.println(**"Something -> AutoCloseableResources\_Second"**);

    }

    @Override

**public** **void** close() **throws** Exception {

        System.out.println(**"Closed AutoCloseableResources\_Second"**);

    }

}

|  |
| --- |
| **private** **void** orderOfClosingResources() **throws** Exception {  **try** (AutoCloseableResourcesFirst af = **new** AutoCloseableResourcesFirst();          AutoCloseableResourcesSecond as = **new** AutoCloseableResourcesSecond()) {            af.doSomething();          as.doSomething();      }  } |

**Output:**

Constructor -> AutoCloseableResources\_First  
Constructor -> AutoCloseableResources\_Second  
Something -> AutoCloseableResources\_First  
Something -> AutoCloseableResources\_Second  
Closed AutoCloseableResources\_Second  
Closed AutoCloseableResources\_First

The try-catch-finally will still work with this feature.