Java Collections Framework + Generics, Lambdas & Stream API

How to be prepared for the interview?

I gathered for you the most popular questions during the Java Interview in JAVA COLLECTIONS FRAMEWORK topic. How you can be prepared for the interview?

Just read the question first. Try to answer the question by yourself. After that, compare your answer with the answer provided. Remember, that this is just a reference to the answer. Because sometimes the topic is big, and you can go really deep with your answer. In case you feel like you don’t understand what the answer is about - feel free to get back in the course and review the relevant section, and relevant lesson one more time.

Also, you are always welcome to ask your questions and I will be happy to answer. I’m sure that these questions will help you to be prepared for the JAVA COLLECTIONS FRAMEWORK interview. You need to be ready to answer perfectly on these questions.

In case there are questions that you don't know the answers to, please, feel free to check my full and the most complete course "Java From Zero to First Job".

Part 1: Java Collections Framework Interview - Questions and Answers

- **WHAT IS A COLLECTION?**

Collections are containers that support various ways of storing and organizing objects in order to be able to access them efficiently. They are implementations of abstract data structures that support three basic operations:

- adding a new element to the collection;
- removing an element from the collection;
- changing an element in the collection.

- **NAME THE MAIN INTERFACES OF COLLECTIONS AND THEIR IMPLEMENTATIONS.**

Three interfaces extend Collection: List, Set, Queue.

- List - stores ordered elements (may be the same); Has such implementations as LinkedList, ArrayList, Vector, etc.

Vector is synchronized, and therefore in one thread, it works slower than other implementations.

ArrayList has advantage in navigating through the collection.

LinkedList has advantage is in inserting and deleting elements in a collection.

- Set - collections that do not contain duplicate elements. Main implementations: HashSet, TreeSet, LinkedHashSet

TreeSet - arranges elements by their values;

HashSet - Orders the elements by their hash keys, although at first glance it may seem that the elements are stored in random order.

LinkedHashSet - stores elements in the order they were added

- Queue is an interface for implementing a queue in Java. Main implementations: LinkedList, PriorityQueue.

Queues work on the principle of FIFO - first in first out.

- Map is an interface for implementing a so-called map, where elements are stored with their keys. Main implementations: HashTable, HashMap, TreeMap, LinkedHashMap

HashTable - synchronized, deprecated.

HashMap - the order of the elements is calculated by the hash key;

TreeMap - elements are stored in sorted order

LinkedHashMap - elements are stored in insertion order

Keys in Mar cannot be the same!

You can synchronize unsynchronized collections and maps using the Collections.synchronizedMap(myMap) / synchronizedList(myList) class.

- ## HOW IS ARRAYLIST DIFFERENT FROM LINKEDLIST? IN WHAT CASES IS IT BETTER TO USE THE FIRST AND IN WHICH CASES THE SECOND?

The difference lies in the way the data is stored. ArrayList stores data as an array, while LinkedList stores as a list (bidirectional).

Reading and retrieving data from the ArrayList is generally faster because we have access to the elements by index.

With a large number of add and remove operations, LinkedList should be a better choice, because these operations do not have to move parts of the array.

• HOW IS HASHMAP DIFFERENT FROM HASHTABLE?

The HashMap class is very similar in functionality to the Hashtable. The main difference is that the methods of the Hashtable class are synchronized, while the HashMap is not. In addition, the HashMap class, unlike the Hashtable, allows the use of null as keys and values.

The presence of synchronization in the Hashtable reduces the performance of the code that uses this class. Therefore, the JCF (Java Collections Framework, introduced in Java 2) classes, including HashMap, are not synchronized. If you still need synchronization, you can use the methods of the Collections class: Collections.synchronizedMap(map), Collections.synchronizedList(list), or Collections.synchronizedSet(set) to wrap your objects into the synchronized wrapper.

These methods return a synchronized decorator for the passed collection.

Starting with Java 6, JCF has been extended with special collections that support multi-threaded access, such as CopyOnWriteArrayList and ConcurrentHashMap.

• HOW IS ARRAYLIST DIFFERENT FROM VECTOR?

The methods of the Vector class are synchronized while the ArrayList is not.

- **WHY MAP IS NOT A COLLECTION WHILE LIST AND SET ARE A COLLECTION?**

A collection (List and Set) is a collection of some elements (usually instances of the same class). Map is a collection of key-value pairs.

Accordingly, some methods of the Collection interface cannot be used in Map. For example, the remove(Object o) method in the Collection interface is designed to remove an element, while the same remove(Object key) method in the Map interface removes an element by a given key.

So, the way how data is structured in Map and Collection is different. That's why we can't say that Map is a collection.

- **GIVE A DEFINITION TO THE TERM "ITERATOR".**

An iterator is an object that allows you to iterate over the elements of a collection. The Iterator declares interface that is used to iterate over different types of data structures. For example, foreach is implemented using an iterator. One of the key methods of the Collection interface is the Iterator<E> iterator() method. It returns an iterator - that is, an object that implements the Iterator interface. Each concrete implementation knows how to iterate over the specific data structure.

- **HOW TO LOOK AT ALL THE KEYS OF A MAP, CONSIDERING THAT MAP IS NOT ITERABLE?**

Use the keySet() method, which returns a set (Set<K>) of keys.

- **HOW TO LOOK AT ALL THE MAP VALUES, CONSIDERING THAT MAP IS NOT ITERABLE?**

Use the values() method, which returns a collection (Collection<V>) of values.

- **HOW TO LOOK AT ALL KEY-VALUE PAIRS IN A MAP, CONSIDERING THAT MAP IS NOT ITERABLE?**

Use the entrySet() method, which returns a set (Set<Map.Entry<K, V>) of key-value pairs.

Part 2: Java Collections Framework Interview - Questions and Answers

- **COMPARE ENUMERATION AND ITERATOR.**

Both interfaces are designed to traverse collections. The Iterator interface was introduced somewhat later in the Java Collections Framework and is preferred.

The main differences between Iterator and Enumeration are:

- the presence of the remove() method to remove an element from the collection during traversal;
- different method names to improve code readability.

- **HOW ARE ITERABLE AND ITERATOR RELATED TO EACH OTHER?**

The Iterable interface has only one method, iterator() , which returns an iterator of the collection to traverse it.

-

- **HOW MANY MEMORY ITEMS ARE ALLOCATED WHEN ARRAYLIST.ADD() is called?**

If there is enough space in the array to accommodate the new element, no additional memory space is allocated. Otherwise, a new array is created.

A new array is created, the size of which is calculated as multiplying the old size by 1.5 (this is true starting from JDK 1.7, in earlier versions the calculations are different).

- **COMPARE ITERATOR AND LISTITERATOR.**

ListIterator extends the Iterator interface to allow the client to traverse the collection in both directions, modify the collection, and get the iterator's current position.

It is important to remember that the ListIterator does not point to a specific element, but its current position is between the elements returned by the previous() and next() methods. Thus, the modification of the collection is carried out for the last element returned by the previous() and next() methods.

- **WHAT WILL HAPPEN IF I CALL ITERATOR.NEXT() WITHOUT "ASKING" ITERATOR.HASNEXT()?**

If the iterator points to the last element of the collection, then a NoSuchElementException will be thrown, otherwise the next element will be returned.

- **IF I HAVE A COLLECTION AND A GENERATED ITERATOR, WILL THE COLLECTION CHANGE IF I CALL ITERATOR.REMOVE()?**

The iterator.remove() method can only be called after the iterator.next() method has been called at least once, otherwise, an IllegalStateException() will be thrown.

If iterator.next() has been called before, then iterator.remove() will remove the element pointed to by the iterator.

- **IF I HAVE A COLLECTION AND A GENERATED ITERATOR, WILL THE ITERATOR CHANGE IF I CALL COLLECTION.REMOVE(..)?**

The iterator will not change, but the next time its methods are called, a ConcurrentModificationException will be thrown telling you that somebody adjust structure of the collection.

There are fail-fast and fail-safe iterators. Fail-safe iterator will not throw ConcurrentModificationException.

- **IS LINKEDLIST A SINGLE-LINKED, DOUBLE-LINKED, OR FOUR-LINKED LIST?**

Doubly Linked List: Each element of a LinkedList stores a link to the previous and next elements.

- **WHAT IS THE WORST TIME OF THE CONTAIN() METHOD FOR AN ELEMENT THAT IS IN A LINKEDLIST (O(1), O(LOG(N)), O(N), O(N*LOG(N)), O(N*N))?**

O(N). The search time for an element is linearly proportional to the number of elements in the list.

- **WHAT IS THE WORST TIME OF THE CONTAIN() METHOD FOR AN ELEMENT THAT IS IN ARRAYLIST (O(1), O(LOG(N)), O(N), O(N*LOG(N)), O(N*N))?**

O(N). The search time for an element is linearly proportional to the number of elements in the list.

- **WHAT IS THE WORST ADD() TIME FOR LINKEDLIST (O(1), O(LOG(N)), O(N), O(N*LOG(N)), O(N*N))?**

O(N). It is worth noting here that adding an element to the end of the list using the add(value), addLast(value) method and adding to the beginning of the list using addFirst(value) takes O(1) time.

O(N) - will be when adding an element to a sorted list, as well as when adding an element using the add(index, value) method.

- **WHAT IS THE WORST ADD() TIME FOR ARRAYLIST (O(1), O(LOG(N)), O(N), O(N*LOG(N)), O(N*N))?**

O(N). Inserting an element at the end of the list takes O(1) time, but if the array capacity is insufficient, then a new array with an increased size is created and all elements from the old array are copied to the new one.

- **HOW MANY MEMORY ITEMS ARE ALLOCATED WHEN LINKEDLIST.ADD() is called?**

One new instance of the nested Node class is created.

- **ESTIMATE THE MEMORY TO STORAGE ONE BYTE PRIMITIVE IN LINKEDLIST?**

Each LinkedList element stores a link to the previous element, the next element, and a link to the data. For x32 systems, each link is 32 bits (4 bytes). The Node itself is approximately 8 bytes. The size of each object in Java is a multiple of 8, so we get 24 bytes. A primitive of type byte occupies 1 byte of memory, but primitives are packed in the list, respectively, we get another 8 bytes. Thus, in the x32 JVM, about 32 bytes are allocated to store a single byte value in a LinkedList.

For a 64-bit JVM, each link is 64 bits (8 bytes). The calculations are similar.

- **ESTIMATE THE AMOUNT OF MEMORY TO STORAGE ONE PRIMITIVE OF TYPE BYTE IN ARRAYLIST?**

ArrayList is array based. Each element of the array stores a primitive data type - byte, the size of which is 1 byte.

- **I ADD AN ELEMENT IN THE MIDDLE OF LIST: LIST.ADD(LIST.SIZE()/2, NEWELEM). FOR WHOM IS THIS OPERATION SLOWER - FOR ARRAYLIST OR FOR LINKEDLIST?**

For ArrayList:

- checking an array for capacity. If there is not enough capacity, then increase the size of the array and copy all the elements into a new array (O(N));
- copying all elements to the right of the insertion position one position to the right (O(N/2));
- element insertion (O(1)).

For Linked List:

- find insertion position (O(N/2));
- element insertion (O(1)).

In the worst case, inserting in the middle of the list is more efficient for LinkedList. In the rest - most likely for ArrayList, since elements are copied using the System.arraycopy() system method.

- **HOW TO LOOK AT LINKEDLIST ELEMENTS IN REVERSE ORDER WITHOUT USING SLOW GET(INDEX)?**

Use a reverse iterator. To do this, LinkedList has a descendingIterator() method.

Part 3: Java Collections Framework Interview - Questions and Answers

- **CAN DIFFERENT OBJECTS IN MEMORY (REF0 != REF1) HAVE REF0.HASHCODE() == REF1.HASHCODE()?**

Yes they can. The hashCode() method does not guarantee the uniqueness of the returned value.

- **CAN DIFFERENT OBJECTS IN MEMORY (REF0 != REF1) HAVE REF0.EQUALS(REF1) == TRUE?**

Yes they can. To do this, the equals() method must be overridden in the class of these objects.

If the Object.equals() method is used, then for two references x and y, the method will return true if and only if both references point to the same object (i.e. x == y returns true).

- **IF THE CLASS POINT{INT X, Y;} IS "CORRECT" TO IMPLEMENT THE EQUALS METHOD (RETURN REF0.X == REF1.X && REF0.Y == REF1.Y), BUT MAKE THE HASH CODE IN THE FORM OF INT HASHCODE() { RETURN X;}, THEN WILL SUCH POINTS CORRECTLY BE PLACED AND REMOVED FROM HASHSET?**

HashSet uses HashMap to store elements (the object itself is used as a key). When an element is added to the HashMap, the hashcode and position in the array where the new element will be inserted is calculated. All instances of the Point class have the same hashcode, causing the hash table to degenerate into a list. When a collision occurs, a check is made for the presence of such an element in the current list.

If the element is found, its value is overwritten. In our case, for different objects, the equals() method will return false. Accordingly, a new element will be added to the HashSet.

Retrieving the element will also succeed.

But the performance of such code will be poor and the benefits of hash tables will not be used.

- **CAN DIFFERENT REFERENCES TO THE SAME OBJECT IN MEMORY (REF0 == REF1) HAVE REF0.EQUALS(REF1) == FALSE?**

No, he can not. The equals() method must guarantee the property of reflexivity: for any non-null x references, the x.equals(x) method must return true.

- **THERE IS A CLASS POINT{INT X, Y;}. WHY IS A HASHCODE OF THE FORM 31 * X + Y PREFERRED THAN X + Y?**

The multiplier makes the hash value dependent on the order in which the fields are processed, and this gives a much better hash function.

- **EQUALS() GENERATES AN EQUIVALENCE RELATION. WHAT PROPERTIES DOES THIS RELATION HAVE: COMMUTATIVITY, SYMMETRY, REFLEXIVITY, DISTRIBUTION, ASSOCIATION, TRANSITIVITY?**

The equals() method should provide:

- symmetry (for any non-null x and y references, the x.equals(y) method must return true if and only if y.equals(x) returns true);
- reflexivity (for any non-null references x, the x.equals(x) method must return true.);
- transitivity (for any non-null references x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) must also return true).

There are also two more properties: persistence and null inequality.

- **EQUALS MUST CHECK THAT THE (EQUALS(OBJECT THAT)) IS THE SAME TYPE OF THE OBJECT ITSELF. WHAT IS THE DIFFERENCE BETWEEN THIS.GETCLASS() == THAT.GETCLASS() AND THAT INSTANCEOF MYCLASS?**

The instanceof operator compares an object and the specified type. It can be used to test whether a given object is an instance of some class, or an instance of its child class, or an instance of a class that implements a specified interface.

getClass() = ... checks two types for identity.

To correctly implement the contract of the equals() method, you must use exact comparison with getClass().

- **WILL HASHMAP WORK IF ALL KEYS RETURN INT HASHCODE() {RETURN 42;}?**

Yes, it will. But then the hash table degenerates into a linked list and loses its advantages.

- **WHY DID HASHMAP BE ADDED IF HASHTABLE ALREADY HAS BEEN ADDED?**

The HashTable class was introduced in JDK 1.0 and is not part of the Java Collection Framework. The methods of the HashTable class are synchronized, which provides thread safety, but this leads to a decrease in performance, which is why the HashMap class was introduced, the methods of which are not synchronized.

In addition, the HashMap class has some other differences: for example, it allows you to store one null key and many null values.

- **IN THEORY, THERE ARE TWO BASIC HASH TABLE IMPLEMENTATIONS: BASED ON OPEN ADDRESSING AND BASED ON THE CHAINING METHOD. HOW IS HASHMAP IMPLEMENTED? WHY DID TEHY IMPLEMENTED IT IN SUCH WAY (IN YOUR OPINION)? WHAT ARE THE PROS AND CONS OF EACH APPROACH?**

The HashMap class is implemented using the chaining method, i.e. Each array cell has its own linked list, and if a collision occurs, a new element is added to this list.

For the chain method, the fill factor can be greater than 1; with an increase in the number of elements, the performance decreases linearly. Such tables are convenient to use if the number of stored elements is not known in advance, or there can be quite a lot of them, which leads to large values of the fill factor.

Among the methods of open implementation, there are:

- linear probing;
- quadratic probing;
- double hashing.

The main disadvantages of structures with the open addressing method:

- The number of elements in the table cannot exceed the size of the array. As the number of elements in the table increases and the load factor increases, the performance of the structure drops sharply, so rehashing is necessary.
- It is difficult to organize the removal of an element.
- Also, the first two methods of open addressing lead to the problem of primary and secondary grouping.

The main advantage of an open addressable hash table is that there is no overhead for creating and storing list objects. It is also easier to organize object serialization/ deserialization.

• HOW MANY NEW OBJECTS ARE CREATED WHEN YOU ADD A NEW ELEMENT TO A HASHMAP?

One new object of the static nested Entry<K,V> class.

• HOW DOES A HASHMAP WORK WHEN YOU ATTEMPT TO SAVE TWO ELEMENTS IN IT BY KEYS WITH THE SAME HASHCODE BUT FOR WHICH EQUALS == FALSE?

The hashCode value is used to calculate the index of the array cell to which the element will be added. Before adding, a check is made for the presence of already existing elements in this cell. If there are no elements, then it is added. If a collision occurs, the list is iteratively traversed looking for an element with the same key and hash code.

If such an element is found, its value is overwritten and the old one is returned. Since the condition says that the added keys are different, the second element will be added to the beginning of the list.

- **HASHMAP CAN BE TURNED INTO A LIST EVEN FOR KEYS WITH DIFFERENT HASHCODE. HOW IS IT POSSIBLE?**

This is possible if the method that determines the array cell number by hashCode will return the same value.

- **WHAT IS THE WORST TIME OF A GET(KEY) METHOD FOR A KEY THAT IS NOT IN THE TABLE (O(1), O(LOG(N)), O(N), O(N*LOG(N)), O(N*N))?**

O(N). The worst case is searching for a key in a table that is turned into a list, the iteration of the keys of which takes time linearly proportional to the number of stored elements.

- **EXPLAIN THE MEANING OF PARAMETERS IN THE HASHMAP(INT INITIALCAPACITY, FLOAT LOADFACTOR) CONSTRUCTOR.**

- int initialCapacity - the initial size of the HashMap (the number of buckets in the hash table at the time of its creation), the default value is 16.

- float loadFactor - HashMap fill factor. It is equal to the ratio of the number of stored elements in the table to its size. loadFactor - is a measure of filling the table with elements, when the number of values stored by the table is

- exceeded, automatic rehashing occurs. The default value of 0.75 is a good compromise between access time and amount of stored data.

Part 4: Java Collections Framework Interview - Questions and Answers

- **WHAT IS THE DIFFERENCE BETWEEN HASHMAP AND IDENTITYHASHMAP? WHAT IS IDENTITYHASHMAP FOR? HOW CAN IT BE USEFUL TO IMPLEMENT SERIALIZATION OR CLONE?**

IdentityHashMap is a data structure that implements the Map interface but uses reference comparison instead of the equals() method when comparing keys (values). In other words, in IdentityHashMap, two keys k1 and k2 will be considered equal if the condition k1 == k2 is satisfied.

IdentityHashMap does not use the hashCode() method, which uses the System.identityHashCode(Object) method instead.

Another difference (as a consequence) is the better performance of IdentityHashMap compared to HashMap if the latter stores objects with costly equals() and hashCode() methods.

One of the main requirements for using HashMap is immutability of the key, however, this requirement does not apply to IdentityHashMap, which does not use the equals() and hashCode() methods.

According to the documentation, such a data structure can be used to implement serialization/cloning. To perform such algorithms, the program needs to maintain a table with all references to objects that have already been processed. Such a table should not treat unique objects as equal, even if the equals() method returns true.

• WHAT IS THE DIFFERENCE BETWEEN HASHMAP AND WEAKHASHMAP? WHAT IS WEAKHASHMAP FOR?

Before considering WeakHashMap , let me briefly remind you what a WeakReference is.

There are 4 types of references in Java: strong (strong reference), soft (SoftReference), weak (WeakReference) and phantom (PhantomReference). The specifics of each link type are related to the way of how the Garbage Collector works. If the object can only be reached by the WeakReference chain (that is, it is not referenced by strong and soft references), then the object will be marked for deletion.

WeakHashMap is a data structure that implements the Map interface and is based on using a WeakReference to store keys. Thus, the key/value pair will be removed from the WeakHashMap if the key object is no longer strongly referenced.

The following situation can be reviewed as an example of using such a data structure: let's imagine that there are objects that need to be expanded with additional information, while changing the class of these objects is undesirable or impossible. In this case, we add each object to WeakHashMap as a key, and as a value - the desired information. Thus, as long as the object has a strong reference (or soft one), you can check the hash table and extract information. Once the object is removed, the WeakReference for that key will be placed in the ReferenceQueue and then the corresponding entry for that weak reference will be removed from the WeakHashMap.

• COMPARE JAVA.UTIL.QUEUE AND JAVA.UTIL.DEQUE INTERFACES.

According to the documentation Deque ("Dec", Double Ended Queue) is a linear collection that supports the insertion / extraction of elements from both ends. In addition, implementations of the Deque interface can be built according to the FIFO or LIFO principle.

Queue is usually (but not necessarily) built according to the FIFO (First-In-First-Out) principle - accordingly, an element is extracted from the beginning of the queue, an element is inserted - to the end of the queue. This principle is violated, for example, by the priority queue (PriorityQueue), which uses the passed comparator when inserting a new element, or the arrangement of elements is carried out according to natural ordering.

Deque extends Queue.

The implementations of both Deque and Queue generally do not override the equals() and hashCode() methods based on the comparison of the stored elements. Instead, inherited methods of the Object class based on reference comparison are used.

• WHY DOES LINKEDLIST IMPLEMENT BOTH LIST AND DEQUE?

LinkedList allows you to add elements to the beginning and end of the list in constant time, which is well suited for implementing the Deque interface (unlike, for example, ArrayList).

• WHAT IS THE DIFFERENCE BETWEEN THE JAVA.UTIL.ARRAYS AND JAVA.LANG.REFLECT.ARRAY CLASSES?

- java.util.Arrays is a class containing static methods for working with arrays, such as searching through an array and sorting it.
- java.lang.reflect.Array - a class for working with arrays when using reflection. Reflection is a mechanism that allows you to examine data about the program during its execution.

• WHAT IS THE DIFFERENCE BETWEEN THE JAVA.UTIL.COLLECTION AND JAVA.UTIL.COLLECTIONS CLASSES?

The java.util.Collections class contains only static methods for working with collections. They include methods that implement polymorphic algorithms (those algorithms that can be used with different types of data structures), "wrappers" that return a new collection with the specified data structure encapsulated, and some other methods.

java.util.Collection is the root interface of the Java Collections Framework. This interface is mainly used where a high level of abstraction is required, such as in the java.util.Collections class.

• WHAT IS "FAIL-FAST BEHAVIOR"?

Fail-fast behavior means that when an error or condition occurs that could lead to an error, the system immediately stops further work and notifies about it.

In the Java Collections API, iterators can either use fail-fast or fail-safe behavior, or be weakly consistent. An iterator with fail-fast behavior will throw a ConcurrentModificationException if the collection has been modified since it was created, i.e. an element is added or removed (without using the iterator's remove() method). The implementation of this behavior is carried out by counting the number of modifications of the collection (modification count):

- when changing the collection (deleting/adding an element), the counter is incremented;
- when creating an iterator, the current value of the counter is passed to it;
- each time the iterator is accessed, the stored value of the counter is compared with the current value, and if they do not match, an exception is thrown.

Using the fail-fast approach avoids non-deterministic program behavior over time. It's also worth noting that fail-fast behavior cannot be absolutely guaranteed.

- ## IS THERE A SPECIAL CLASS JAVA.UTIL.ENUMSET FOR ENUMS? WHY? WHY DID THE AUTHORS ARE NOT SATISFIED WITH HASHSET OR TREESET?

EnumSet is a variation of the implementation of the Set interface for use with enums (Enum). EnumSet uses an array of bits to store values (bit vector), which allows for high compactness and efficiency. The data structure stores objects of only one Enum type, which is specified when an EnumSet instance is created. All basic operations are performed in constant time (O(1)) and are generally somewhat faster (though not guaranteed) than their counterparts in the HashSet implementation. Bulk operations (like containsAll() and retainAll()) are very fast if their argument is an instance of type Enum.

In addition, the EnumSet class provides many static initialization methods for easy and convenient instantiation.

Iteration over the EnumSet is carried out according to the order in which the enumeration elements are declared.

Part 5: Java Collections Framework Interview - Questions and Answers

- ## WHICH COLLECTION IMPLEMENTS THE FIFO APPROACH?

FIFO - First-In-First-Out (first in, first out). Such a data structure as a queue (java.util.Queue) is usually built according to this principle.

- ## JAVA.UTIL.STACK - CONSIDERED "DEPRECATED". WHAT IS IT RECOMMENDED TO REPLACE IT? WHY?

It is recommended to use the Deque ("Dec", Double Ended Queue) interface and its implementations.

A stack is a data structure based on the LIFO (Last-In-First-Out, or FILO) principle. Each new value is added to the "top" of the stack, and the last added element (from the "top" of the stack) is retrieved. When an element is retrieved, it is removed from the data structure.

The Stack class appeared in JDK 1.0 and extends the Vector class, inheriting its functionality, which somewhat violates the concept of a stack (for example, the Vector class provides the ability to access any element by index). Also, the use of Deque allows you to follow the principle of programming at the level of interfaces, rather than specific implementations, which facilitates further support of the developed class and increases its flexibility, allowing you to change the deque implementation to the desired one if necessary.

- ## LINKEDHASHMAP - WHAT IS ANOTHER "BEAST"? WHAT'S IN IT FROM LINKEDLIST AND WHAT'S FROM HASHMAP?

The implementation of LinkedHashMap differs from HashMap by supporting a doubly linked list that determines the order of iteration over the elements of the data structure. By default, the elements of the list are ordered according to their insertion order in the LinkedHashMap (insertion-order). However, the iteration order can be changed by setting the accessOrder constructor parameter to true. In this case, access is carried out in the order of the last access to the element (access-order). This means that when the get() or put() methods are called, the element being accessed is moved to the end of the list.

When adding an element that is already present in the LinkedHashMap (i.e. with the same key), the iteration order over the elements does not change.

- ## CAN WE MAKE A SIMPLE CACHE WITH "INVALIDATION POLICY" ON LINKEDHASHMAP. DO YOU KNOW HOW?

You need to use the LRU algorithm (Least Recently Used algorithm) and LinkedHashMap with access-order. In this case, when an element is accessed, it will move to the end of the list, and the least used elements will be gradually grouped at the beginning of the list.

To do this, the standard implementation of LinkedHashMap (source) has a method

removeEldestEntries() which returns true if the current object is a LinkedHashMap

should remove the least used element from the collection. The method is called when using the put() and putAll() methods.

To implement a simple example of a cache with clearing old values when a specified threshold is exceeded we need to declare a new class and extend it from the LinkedHashMap. After that, we need to override removeEldestEntry() method in order to return true when size reaches max allowed entries. Max allowed entries can be configured during the cache instantiation.

It is worth noting that LinkedHashMap does not allow to fully implement the LRU algorithm, since when inserting an element already in the collection, the iteration order does not change.

- ## LINKEDHASHSET - WHAT IS ANOTHER "BEAST"? WHAT'S IN IT FROM LINKEDLIST AND WHAT'S FROM HASHSET?

The implementation of LinkedHashSet differs from HashSet by supporting a doubly linked list that determines the order of iteration over the elements of the data structure. The elements of the list are ordered according to their insertion order in the LinkedHashSet (insertion-order).

When adding an element that is already present in the LinkedHashSet (i.e. with the same key), the iteration order over the elements does not change.

- ## WHAT DOES PRIORITYQUEUE DO?

PriorityQueue is a data structure that arranges elements in natural order, or using a Comparator passed to the constructor.

We can use it to store objects according to their priority: for example, sorting doctor's patients - emergency patients are moved to the front of the queue, less urgent patients - closer to the end of the queue.

- ## WHAT IS JAVA.UTIL.COMPARATOR DIFFERENT FROM JAVA.LANG.COMPARABLE?

Interface Comparable sets a comparison property to the object that implements it. That is, it makes the object comparable (according to the developer's rules).

Interface Comparator allows you to create objects that will control the comparison process (for example, when sorting).

How to contact me 👇

Linkedin: https://www.linkedin.com/mynetwork/
GitHub: https://github.com/Hugs-4-Bugs
HackerRank : https://www.hackerrank.com/Prabhat_7250
GeeksforGeeks: https://auth.geeksforgeeks.org/user/prabhatkueazc/practice
LeetCode: https://leetcode.com/kattyprabhat/
Instagram: https://www.instagram.com/_s_4_sharma/?hl=en
Twitter: https://twitter.com/kattyPrabhat
Gmail: prabhatkumarssm72@gmail.com

REFERENCES :

- GeeksforGeeks
- InterviewBit
- w3School
- Javatpoint
- Tutorialspoint
- Google