**Java Interview Questions**

**1-Sort map based on values**

**public** **class** SortMapByValues {

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

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

hm.put("Math", 98);

hm.put("Data Structure", 85);

hm.put("Database", 91);

hm.put("Java", 95);

Map<String, Integer> hm1 = *sortByValueLambdaSteam*(hm);

**for** (Map.Entry<String, Integer> en : hm1.entrySet()) {

System.***out***.println("Key = " + en.getKey() + ", Value = " + en.getValue());

}

}

**public** **static** HashMap<String, Integer> sortByValueOld(HashMap<String, Integer> hm) {

// Create a list from elements of HashMap

List<Map.Entry<String, Integer>> list = **new** LinkedList<Map.Entry<String, Integer>>(hm.entrySet());

// Sort the list

Collections.*sort*(list, **new** Comparator<Map.Entry<String, Integer>>() {

**public** **int** compare(Map.Entry<String, Integer> obj1, Map.Entry<String, Integer> obj2) {

**return** (obj1.getValue()).compareTo(obj2.getValue());

}

});

// put data from sorted list to LinkedHashMap so insertion order is preserved

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

**for** (Map.Entry<String, Integer> currentEntry : list) {

valueSortedMap.put(currentEntry.getKey(), currentEntry.getValue());

}

**return** valueSortedMap;

}

**public** **static** HashMap<String, Integer> sortByValueLambda(HashMap<String, Integer> inputMap) {

// Create a list from elements of HashMap

List<Map.Entry<String, Integer>> list = **new** LinkedList<Map.Entry<String, Integer>>(inputMap.entrySet());

// Sort the list using lambda expression

Collections.*sort*(list, (i1, i2) -> i1.getValue().compareTo(i2.getValue()));

// put data from sorted list to HashMap

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

**for** (Map.Entry<String, Integer> aa : list) {

valueSortedMap.put(aa.getKey(), aa.getValue());

}

**return** valueSortedMap;

}

**public** **static** HashMap<String, Integer> sortByValueLambdaSteam(HashMap<String, Integer> hm) {

HashMap<String, Integer> valueSortedMap = hm.entrySet()

.stream()

.sorted((i1, i2) -> i1.getValue().compareTo(i2.getValue()))

.collect(Collectors.*toMap*(

Map.Entry::getKey,

Map.Entry::getValue,

(e1, e2) -> e1,

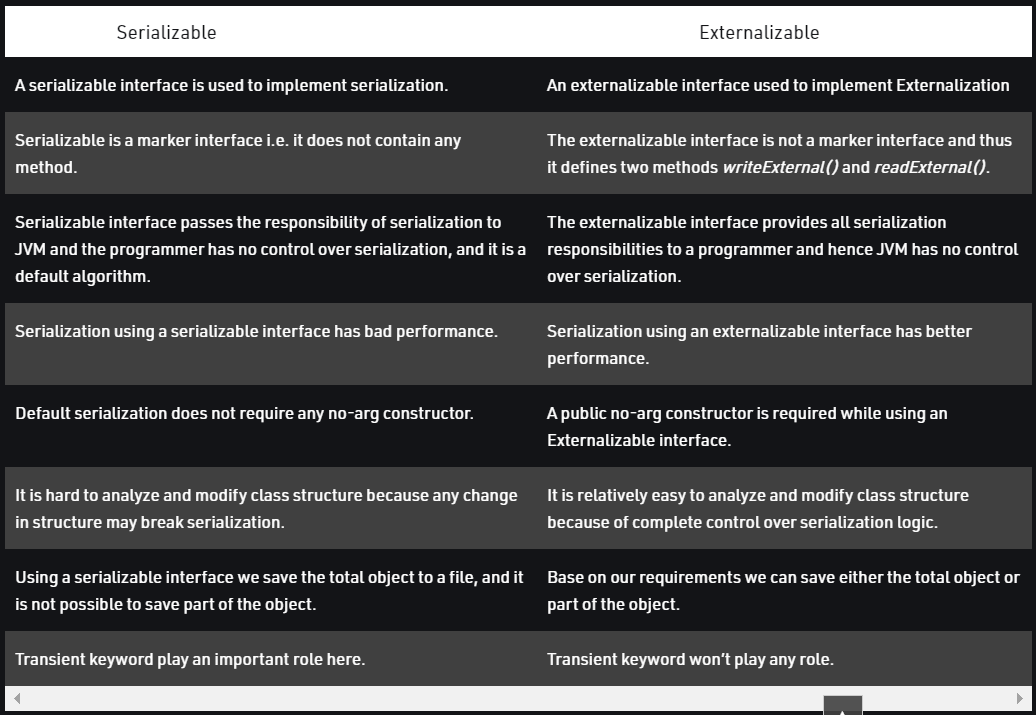
LinkedHashMap::**new**));

**return** valueSortedMap;

}

}

**2-Serialization vs Externalization**



The process of writing the state of an object to a file is called serialization, but strictly speaking, it is the process of converting an object from java supported form into a file supported form or network supported form. By using fileOutputStream and objectOutputStream classes we can implement serialization.

**class** serializableDemo **implements** Serializable {

String name;

**int** age;

**int** jobId;

// Default constructor

**public** serializableDemo(String name, **int** age, **int** jobId) {

**this**.name = name;

**this**.age = age;

**this**.jobId = jobId;

}

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

// Java Object

serializableDemo t1 = **new** serializableDemo("Ram", 34, 2364);

// Serialization -> Saving of object in a file

FileOutputStream fos = **new** FileOutputStream("abc1.ser");

ObjectOutputStream oos = **new** ObjectOutputStream(fos);

oos.writeObject(t1);

System.***out***.println("Object has been serialized");

//Deserialization -> Reading the object from a file

FileInputStream fis = **new** FileInputStream("abc1.ser");

ObjectInputStream ois = **new** ObjectInputStream(fis);

serializableDemo t2 = (serializableDemo) ois.readObject();

System.***out***.println("Object has been deserialized ");

//Printing Deserialized Object

System.***out***.println("Name:" + t2.name + "\n"

+ "Age:" + t2.age + "\n"

+ t2.jobId);

}

}

**public** **class** ExternalizableDemo **implements** Externalizable {

String name;

**int** age;

**int** jobId;

// No-argument constructor

**public** ExternalizableDemo() {

System.***out***.println("Public no-argument constructor");

}

// Default constructor

**public** ExternalizableDemo(String name, **int** age, **int** jobId) {

**this**.name = name;

**this**.age = age;

**this**.jobId = jobId;

}

// Implementing write external method

**public** **void** writeExternal(ObjectOutput out) **throws** IOException {

out.writeObject(name);

out.writeInt(age);

}

// Implementing readExternal method

**public** **void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {

name = (String) in.readObject();

age = in.readInt();

}

// Main method

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

// Java Object

ExternalizableDemo t1 = **new** ExternalizableDemo("Ram", 35, 23675);

// Serialization -> Saving of object in a file

FileOutputStream fos = **new** FileOutputStream("abc.ser");

ObjectOutputStream oos = **new** ObjectOutputStream(fos);

oos.writeObject(t1);

// Deserialization -> Reading the object from a file

FileInputStream fis = **new** FileInputStream("abc.ser");

ObjectInputStream ois = **new** ObjectInputStream(fis);

ExternalizableDemo t2 = (ExternalizableDemo) ois.readObject();

// Printing Deserialized Object

System.***out***.println("Name :"

+ " " + t2.name + " "

+ "Age :"

+ " " + t2.age);

}

}

**3-Transient vs volatile**

A volatile keyword is used in a multithreading environment where two threads reading and writing the same variable simultaneously. The volatile keyword flushes the changes directly to the main memory instead of the CPU cache (i.e. in the thread stack).

On the other hand, the transient keyword is used during serialization. Fields that are marked as transient cannot be part of the serialization and deserialization. We don't want to save the value of any variable then we use transient keyword with that variable.

| **Sr. No.** | **Key** | **Volatile** | **Transient** |
| --- | --- | --- | --- |
| 1 | Basic | Volatile keyword is used to flush changes directly to the main memory | The transient keyword is used to exclude variable during serialization |
| 2. | Default value | Volatile are not initialized with a default value. | During deserialization, transient variables are initialized with a default value |
| 3 | Static | Volatile can be used with a static variable. | Transient cannot be used with the static keyword |
| 4 | Final | Volatile can be used with the final keyword | Transient cannot be used with the final keyword |

// A sample class that uses transient keyword to skip their serialization.

**class** TransientExample **implements** Serializable {

**transient** **int** age;

// serialize other fields

**private** String name;

**private** String address;

// other code

}

**public** **class** VolatileExmaple **extends** Thread {

**volatile** **boolean** isRunning = **true**;

@Override

**public** **void** run() {

**long** count = 0;

**while** (isRunning) {

count++;

}

System.***out***.println("Thread terminated with count= " + count);

}

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

VolatileExmaple t = **new** VolatileExmaple();

t.start();

Thread.*sleep*(2000);

t.isRunning = **false**;

t.join();

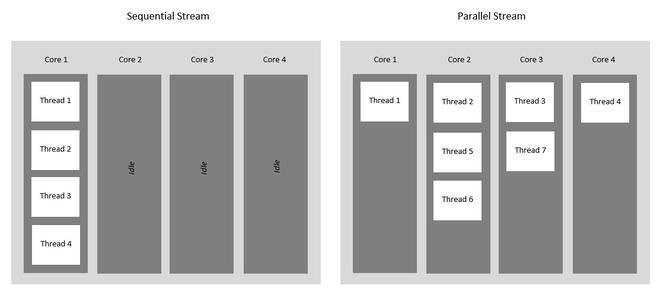
System.***out***.println("isRunning set to " + t.isRunning);

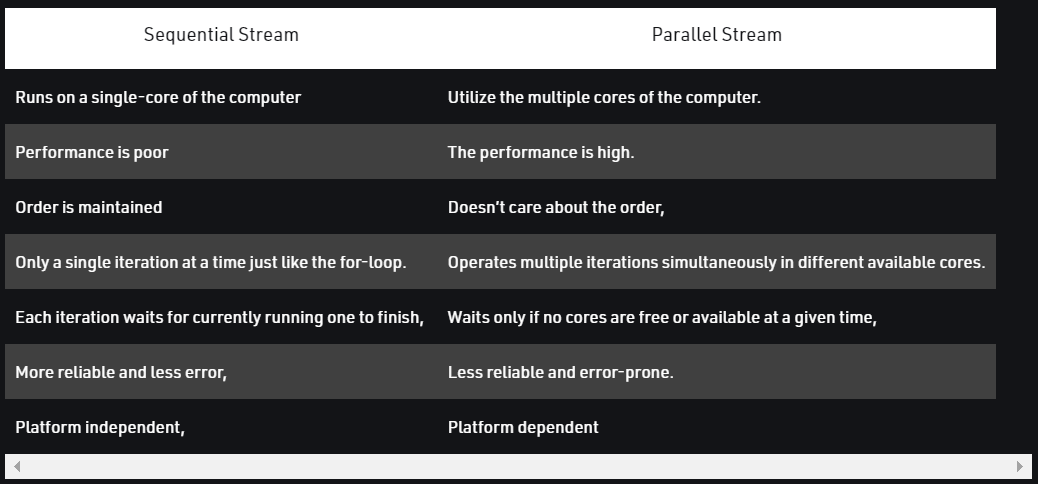
}

}

**4-Stream vs parallel stream**

A stream in Java is a sequence of objects which operates on a data source such as an array or a collection and supports various methods. It was introduced in Java 8’s java.util.stream package. Stream supports many aggregate operations like filter, map, limit, reduce, find, and match to customize the original data into a different form according to the need of the programmer. The operations performed on a stream do not modify its source hence a new stream is created according to the operation applied to it. The new data is a transformed copy of the original form.



****

**class** SequentialStreamDemo {

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

// create a list

List<String> list = Arrays.*asList*("Hello ", "G", "E", "E", "K", "S!");

list.stream().forEach(System.***out***::print);

}

}

Output: -

**Hello GEEKS!**

**class** ParallelStreamExample {

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

// create a list

List<String> list = Arrays.*asList*("Hello ", "G", "E", "E", "K", "S!");

**list.parallelStream().forEach(System.*out*::print);**

}

}

Output: -

**EGS!KEHello**

**class** ParallelStreamWithOrderedIteration {

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

// create a list

List<String> list = Arrays.*asList*("Hello ", "G", "E", "E", "K", "S!");

**list.parallelStream().forEachOrdered(System.*out*::print);**

}

}

Output: -

**Hello GEEKS!**