**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,

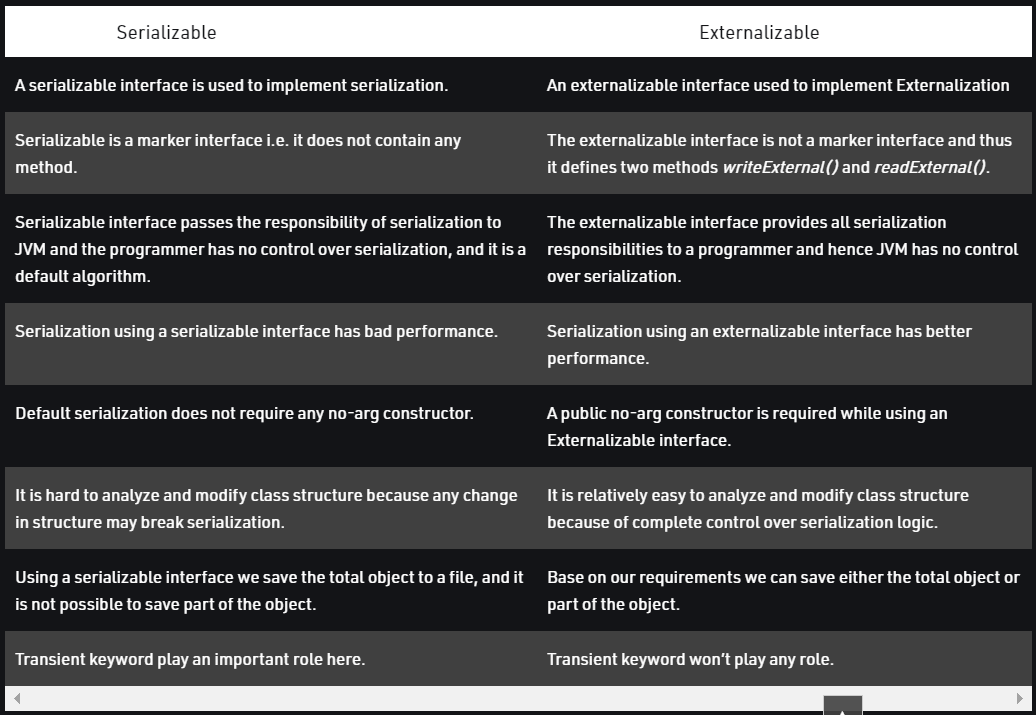
LinkedHashMap::**new**));

**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();

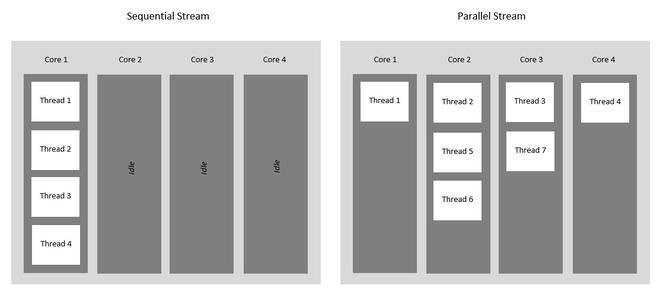
System.***out***.println("isRunning set to " + t.isRunning);

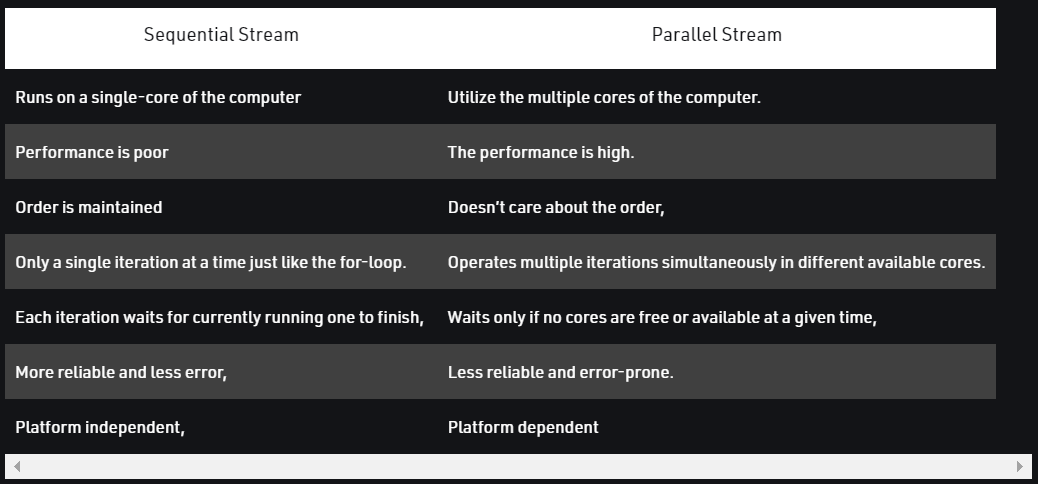
}

}

**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!**

**5-Java is pass by value or pass by reference?**

Data is shared between functions by passing parameters. Now, there are 2 ways of passing parameters:

1. Pass By Value: The pass by value method copies the value of actual parameters. The called function creates its own copy of argument values and then uses them. Since the work is done on a copy, the original parameter does not see the changes.
2. Pass By Reference: The pass by reference method passes the parameters as a reference(address) of the original variable. The called function does not create its own copy, rather, it refers to the original values only. Hence, the changes made in the called function will be reflected in the original parameter as well.

Java follows the following rules in storing variables:

1. Local variables like primitives and object references are created on Stack memory.
2. Objects are created on Heap memory.

Now coming to the main question: Is Java Pass by Value or Pass by Reference?

**Java Always follows Pass by Value**

**public** **class** PassByValueReference {

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

**int** data = 10;

List<String> fruits = **new** ArrayList<>();

fruits.add("apple");

fruits.add("mango");

fruits.add("grapes");

System.***out***.println("Before processing pass by value");

System.***out***.println(data);

System.***out***.println("After processing pass by value");

System.***out***.println(data);

System.***out***.println("Before processing pass by ref");

System.***out***.println(fruits);

System.***out***.println("After processing pass by ref");

*processPassByRef*(fruits);

System.***out***.println(fruits);

}

**public** **static** **void** processPassByValue(**int** data) {

data = data \* 10;

}

**public** **static** **void** processPassByRef(List<String> fruitsRef) {

fruitsRef.add("Banana");

}

}

Before processing pass by value

10

After processing pass by value

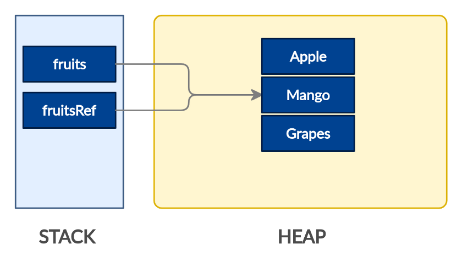
10

Before processing pass by ref

[apple, mango, grapes]

After processing pass by ref

[apple, mango, grapes, Banana]



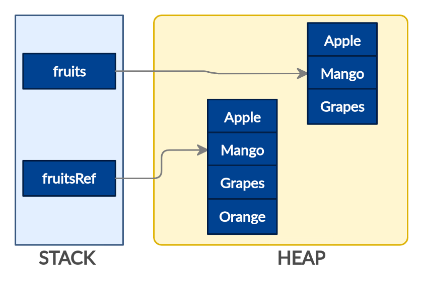
**In Below case:**

**public** **static** **void** processPassByRef(List<String> fruitsRef) {

fruitsRef = **new** ArrayList<>();

fruitsRef.add("Banana");

}



**In the above program, “fruits” is passed to the processData function. “fruitsRef” is a copy of the “fruits” param. Both fruits and fruitsRef are created on Stack. They are two different references. But the interesting point is, it points to the same underlying object in Heap. So, any change that you make using one reference is going to impact the common object.**

**6- Employees Sorting question**

**public** **class** EmployeeSortingQuestion {

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

Employee emp1 = **new** Employee(1, "ghi");

Employee emp2 = **new** Employee(2, "def");

Employee emp3 = **new** Employee(3, "abc");

Employee emp4 = **new** Employee(4, "xyz");

List<Employee> empList = **new** ArrayList<>();

empList.add(emp1);

empList.add(emp2);

empList.add(emp3);

empList.add(emp4);

System.***out***.println("Before Sorting");

System.***out***.println(empList);

System.***out***.println("After EmpNameSorter");

Collections.*sort*(empList, **new** EmpNameSorter());

System.***out***.println(empList);

System.***out***.println("After Java8 EmployeeSorting");

empList.sort(Comparator.*comparing*(e -> e.getEmpName()));

System.***out***.println(empList);

System.***out***.println("After Java8 EmployeeSorting reversed");

Comparator<Employee> empReverseComparator = Comparator.*comparing*(e -> e.getEmpName());

empList.sort(empReverseComparator.reversed());

System.***out***.println(empList);

}

}

**class** EmpNameSorter **implements** Comparator<Employee> {

@Override

**public** **int** compare(Employee obj1, Employee obj2) {

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

}

}

**class** Employee {

**int** empId;

String empName;

**public** Employee(**int** empId, String empName) {

**super**();

**this**.empId = empId;

**this**.empName = empName;

}

**public** **int** getEmpId() {

**return** empId;

}

**public** **void** setEmpId(**int** empId) {

**this**.empId = empId;

}

**public** String getEmpName() {

**return** empName;

}

**public** **void** setEmpName(String empName) {

**this**.empName = empName;

}

@Override

**public** String toString() {

**return** "[empId=" + empId + "empName=" + empName +"]";

}

}

Before Sorting

[[empId=1empName=ghi], [empId=2empName=def], [empId=3empName=abc], [empId=4empName=xyz]]

After EmpNameSorter

[[empId=3empName=abc], [empId=2empName=def], [empId=1empName=ghi], [empId=4empName=xyz]]

After Java8 EmployeeSorting

[[empId=3empName=abc], [empId=2empName=def], [empId=1empName=ghi], [empId=4empName=xyz]]

After Java8 EmployeeSorting reversed

[[empId=4empName=xyz], [empId=1empName=ghi], [empId=2empName=def], [empId=3empName=abc]]

**7- Remove duplicates in ArrayList of Employees**

**public** **class** RemoveArrayListDuplicates {

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

List<Employee> employeeList = **new** ArrayList<>();

Employee emp1 = **new** Employee(1, "abc");

Employee emp2 = **new** Employee(2, "def");

Employee emp3 = **new** Employee(1, "abc");

employeeList.add(emp1);

employeeList.add(emp2);

employeeList.add(emp3);

System.***out***.println(employeeList);

List<Employee> listWithoutDuplicates = **new** ArrayList<>(**new** HashSet<>(employeeList));

System.***out***.println(listWithoutDuplicates);

}

}

[[empId=1empName=abc], [empId=2empName=def], [empId=1empName=abc]]

[[empId=1empName=abc], [empId=2empName=def]]

Note: hashCode() and equals() should be implemented in Employee class if this should work.

**8- Singleton Class**

**public** **class** SingletonClass **implements** Serializable, Cloneable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

// Lazy initialization

**private** **static** SingletonClass *singleInstance* = **null**;

// Prevent from Reflection.

**private** SingletonClass() {

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

**throw** **new** InstantiationError("Creating of this object is not allowed.");

}

}

// Use of double locking.

**public** **static** SingletonClass getInstance() {

**if** (*singleInstance* == **null**) {

**synchronized** (SingletonClass.**class**) {

**if** (*singleInstance* == **null**) {

*singleInstance* = **new** SingletonClass();

}

}

}

**return** *singleInstance*;

}

// Clone prevention

@Override

**protected** Object clone() **throws** CloneNotSupportedException {

**return** *getInstance*();

}

// Deserialization prevention

**protected** Object readResolve() {

**return** *getInstance*();

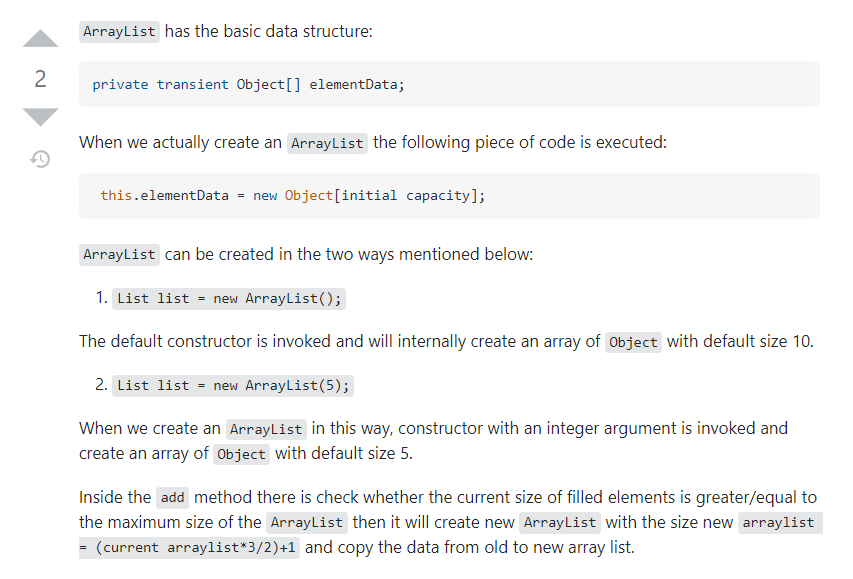
}

}

**Double Locking:**

Here, we run into a problem. Suppose that there are two threads running. Both can get inside of the if statement concurrently when the instance is null. Then, one thread enters the synchronized block to initialize the instance, while the other is blocked. When the first thread exits in the synchronized block, the waiting thread enters and creates another singleton object. Note that when the second thread enters the synchronized block, it does not check to see if the instance is non-null.

**9-Internal working of ArrayList**

****

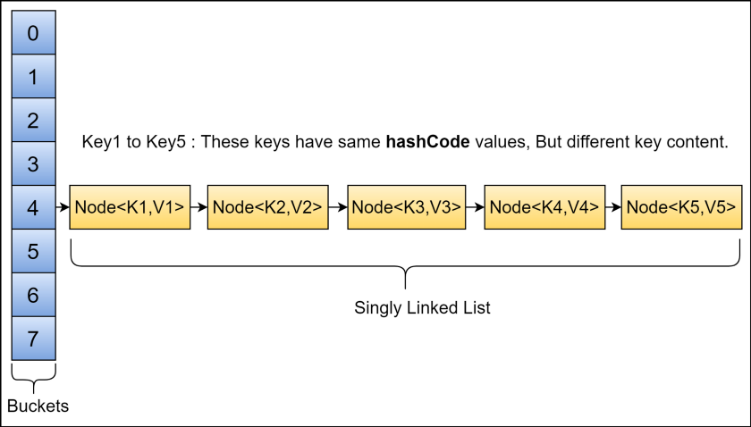
**10- Internal working of HashMap**

***Few important points to about HashMap:***

1. HashMap uses its static inner class Node<K,V> for storing the entries into the map.
2. HashMap allows at most one null key and multiple null values.
3. The HashMap class does not preserve the order of insertion of entries into the map.
4. HashMap has multiple buckets or bins which contain a head reference to a singly linked list. That means there would be as many linked lists as there are buckets. Initially, it has a bucket size of 16 which grows to 32 when the number of entries in the map crosses the 75%. (That means after inserting in 12 buckets bucket size becomes 32)
5. HashMap is almost similar to Hashtable except that it’s unsynchronized and allows at max one null key and multiple null values.
6. HashMap uses hashCode() and equals() methods on keys for the get and put operations. So HashMap key objects should provide a good implementation of these methods.
7. That’s why the Wrapper classes like Integer and String classes are a good choice for keys for HashMap as they are immutable and their object state won’t change over the course of the execution of the program.

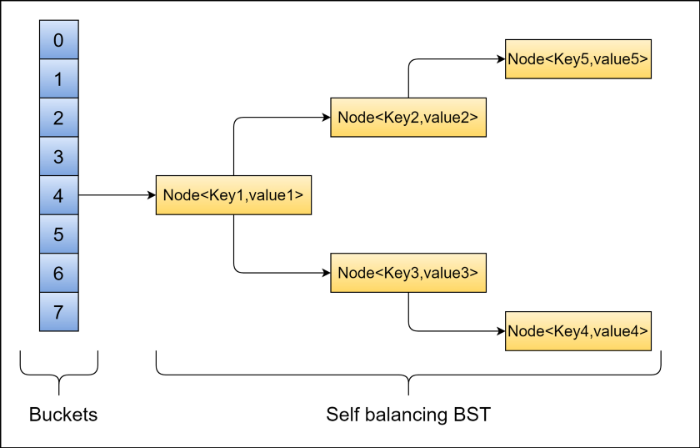
***Now Let’s Look at the Internal Working of HashMap:***

* HashMap uses its static inner class Node<K,V> for storing map entries. That means each entry in hashMap is a Node. Internally HashMap uses a hashCode of the key Object and this hashCode is further used by the hash function to find the index of the bucket where the new entry can be added.
* HashMap uses multiple buckets and each bucket points to a Singly Linked List where the entries (nodes) are stored.
* Once the bucket is identified by the hash function using hashcode, then hashCode is used to check if there is already a key with the same hashCode or not in the bucket (singly linked list).
* If there already exists a key with the same hashCode, then the equals() method is used on the keys. If the equals method returns true, that means there is already a node with the same key and hence the value against that key is overwritten in the entry(node), otherwise, a new node is created and added to this Singly Linked List of that bucket.
* If there is no key with the same hashCode in the bucket found by the hash function then the new Node is added into the bucket found.



***A Very Important Note That You Must Know:***

Before java 8, singly-linked lists were used for storing the nodes. But this implementation has changed to self-balancing BST after a threshold is crossed (static final int TREEIFY\_THRESHOLD = 8;). The motive behind this change is that HashMap buckets normally use linked lists, but for the linked lists the worst-case time is O(n) for lookup. Also note that Ordinary binary search trees have pathological cases where they become O(n) [basically BST becomes skewed], but red-black/AVL trees are specifically designed to prevent these cases. In a HashMap with linked lists, if we have a really awful hash function, we could end up with all the items hashing to the same bucket and get O(n) lookup, But it seems like with this red-black/AVL tree scheme, even if all the items hashed into the same bucket, we would get O(log n) lookup in worst of worst scenario.



**Re-Hashing:**

Whenever the number of entries in the HashMap crosses the threshold value then the bucket size of the HashMap is doubled and rehashing is performed and all the already existing entries of the map are copied and new entries are added to this increased HashMap.

Threshold value = Bucket size \* Load factor

Eg. If bucket size is 16 and the load factor is 0.75 then the threshold value is 12.

**Time Complexity:**

1. In a fairly distributed HashMap where the entries go to all the buckets in such a scenario, the HashMap has O(1) time for search, insertion, and deletion operations.
2. In the worst case, where all the entries go to the same bucket and the singly linked list stores these entries, O (n) time is required for operations like search, insert, and delete.
3. In a case where the threshold for converting this linked list to a self-balancing binary search tree (i.e., AVL/Red black) is used then for the operations, search, insert and delete O(log N) is required as AVL/Red Black tree has a max length of log N in the worst case.

**11- Why HashMap key should be immutable?**

1. If key's hash code changes after the key-value pair (Entry) is stored in HashMap, the map will not be able to retrieve the Entry.
2. **Key's hashcode can change if the key object is mutable.**
3. **Mutable keys in HashMap can result in data loss.**
4. String, Integer and other wrapper classes are natural candidates of HashMap key, and String is most frequently used key as well because String is immutable and final, and overrides equals and hashcode() method .

**12- Use of private constructor in java**

1. While implementing Singleton class.
2. If you have a utility class that generally has only public static methods and you do not want to create an object to that utility class.

**13- AutoClosable in java**

1. A resource can be thought of as something controlled by the operating system, a file, byte array, buffered reader or a socket for example, that can be accessed and borrowed by a program. It is important for the program that accesses the resource to return it to the operating system once it has finished using it. **An Object that implements the AutoCloseable interface holds on to a resource until it is done using it within a try block and then it is automatically closed. Whereas an object that implements a Closable interface can be closed by calling close(). Both AutoCloseable and Closable only include one abstract method close() which closes a resource and releases any underlying resources associated with it.**
2. If you don’t close the resources you are using, eventually you will run out of them. For example, if you don’t close a Socket, that port the Socket is running on cannot be used and there are a finite number of ports that a Socket can use on any machine. 65535 ports to be precise. Closing resources is not always handled by a garbage collector and so can lead to memory wastage. Not good.

**public** **class** AutoClosableExample {

**public** String readBlogDraftWithoutAutoClosable(String path) **throws** IOException {

BufferedReader buffer = **new** BufferedReader(**new** FileReader(path));

**try** {

**return** buffer.readLine();

} **finally** {

buffer.close();

}

}

//Try with resources

**public** String readBlogDraftWithAutoClosable(String path) **throws** IOException {

**try** (BufferedReader buffer = **new** BufferedReader(**new** FileReader(path))) {

**return** buffer.readLine();

}

}

}

***Here, BufferedReader extends Reader and Reader implements Closable. And Closable extends AutoClosable.***

**14- Marker Interface**

An interface that does not contain methods, fields, and constants is known as marker interface. In other words, **an empty interface is known as marker interface or tag interface**. It delivers the run-time type information about an object**. It is the reason that the JVM and compiler have additional information about an object**. The **Serializable** and **Cloneable** interfaces are the example of marker interface. In short, it indicates a signal or command to the JVM.

There are the two alternatives of marker interface that produces the same result as the marker interface.

1. **Internal Flags**: It can be used in place of marker interface to indicate any specific operation.
2. **Annotations**: Since Java 5, marker interfaces are omitted. Instead of marker interface, Java 5 provides the annotations to achieve the same results. It allows flexible metadata capability. Therefore, by applying annotations to any class, we can perform specific action.

**Uses of Marker Interface**

Marker interface is used as a tag that inform the Java compiler by a message so that it can add some special behavior to the class implementing it.

**Built-in Marker Interfaces**

1. Cloneable Interface
2. Serializable Interface
3. Remote Interface (Remote interface is a marker interface that belong to java.rmi package. **It marks an object as remote that can be accessed from another machine (host).**)

//custom marker interface

**interface** Car {

}

**class** Vehicle **implements** Car {

**static** **void** isVehicle() {

System.***out***.println("Car is a vehicle.");

}

}

**public** **class** CustomMarkerInterfaceExample {

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

Vehicle.*isVehicle*();

}

}

**15- Garbage Collection in java**

When a program executes in Java, it uses memory in different ways. The heap is a part of memory where objects live. It's the only part of memory that involved in the garbage collection process. It is also known as garbage collectible heap. **All the garbage collection makes sure that the heap has as much free space as possible. The function of the garbage collector is to find and delete the objects that cannot be reached.**

1. It is controlled by a thread known as Garbage Collector.
2. Java provides two methods System.gc() and Runtime.gc() that sends request to the JVM for garbage collection. Remember, it is not necessary that garbage collection will happen.
3. Java programmer are free from memory management. We cannot force the garbage collector to collect the garbage, it depends on the JVM.
4. If the Heap Memory is full, the JVM will not allow to create a new object and shows an error java.lang.OutOfMemoryError.
5. When garbage collector removes object from the memory, first, the garbage collector thread calls the finalize() method of that object and then remove

All the objects which are created dynamically (using new in C++ and Java) are allocated memory in the heap. **If we go on creating objects, we might get Out of Memory error, since it is not possible to allocate heap memory to objects. So, we need to clear heap memory by releasing memory for all those objects which are no longer referenced by the program (or the unreachable objects) so that the space is made available for subsequent new objects.** This memory can be released by the programmer itself but it seems to be an overhead for the programmer, here garbage collection comes to our rescue, and it automatically releases the heap memory for all the unreferenced objects.

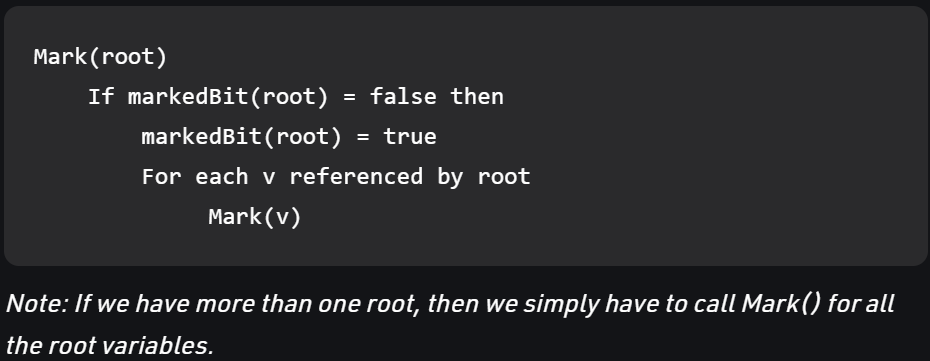
There are many garbage collection algorithms which run in the background. One of them is mark and sweep.

The Mark and Sweep Algorithm occurs in two phases:

1. Mark phase
2. Sweep phase

**Mark Phase**

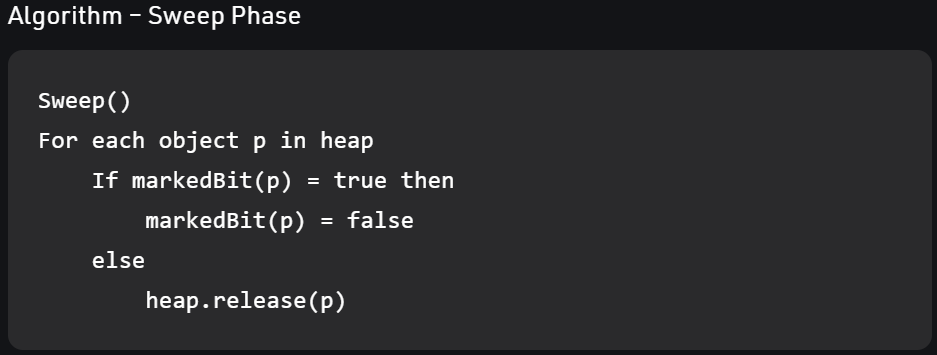
When an object is created, its mark bit is set to 0(false). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). Now to perform this operation we simply need to do a graph traversal; a **depth first search approach** would work for us. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.

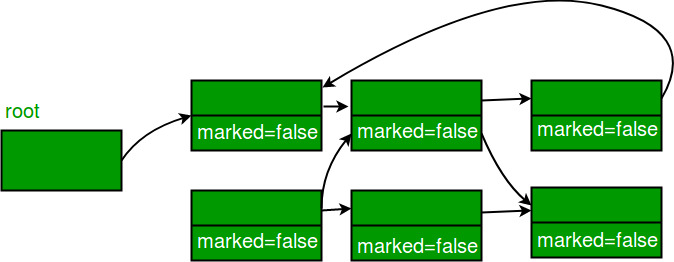


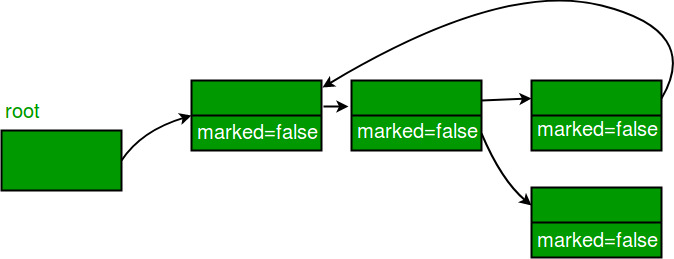
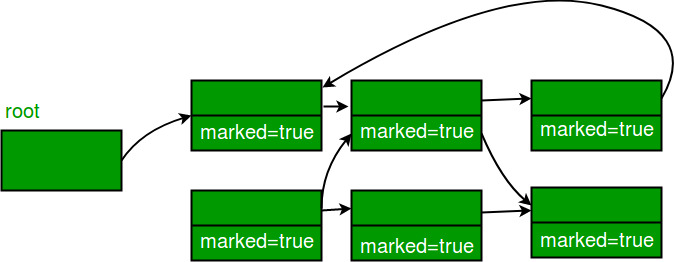
**Sweep Phase**

As the name suggests it “sweeps” the unreachable objects i.e. it clears the heap memory for all the unreachable objects. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to true.

Now the mark value for all the reachable objects is set to false, since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects.







**Advantages of Mark and Sweep Algorithm**

1. It handles the case with cyclic references, even in case of a cycle, this algorithm never ends up in an infinite loop.
2. There are no additional overheads incurred during the execution of the algorithm.

**Disadvantages of Mark and Sweep Algorithm**

1. The main disadvantage of the mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs.

**16- Types of Garbage Collectors in java**

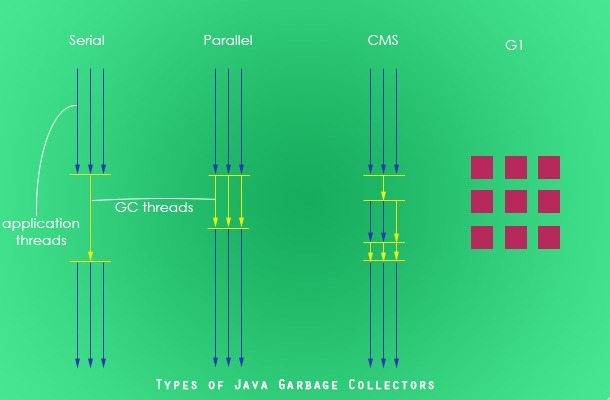


1. Serial Garbage Collector
2. Parallel Garbage Collector
3. CMS Garbage Collector
4. G1 Garbage Collector

Each of these four types has its own advantages and disadvantages. Most importantly, we the programmers can choose the type of garbage collector to be used by the JVM. We can choose them by passing the choice as JVM argument. Each of these types differ largely and can provide completely different application performance. It is critical to understand each of these types of garbage collectors and use it rightly based on the application.

**Serial garbage collector** works by holding all the application threads. It is designed for the single-threaded environments. It uses just a single thread for garbage collection. The way it works by freezing all the application threads while doing garbage collection may not be suitable for a server environment. It is best suited for simple command-line programs.

**Parallel garbage collector** is also called as throughput collector. It is the default garbage collector of the JVM. Unlike serial garbage collector, this uses multiple threads for garbage collection. Similar to serial garbage collector this also freezes all the application threads while performing garbage collection.



**Concurrent Mark Sweep (CMS)** garbage collector uses multiple threads to scan the heap memory to mark instances for eviction and then sweep the marked instances. CMS garbage collector holds all the application threads in the following two scenarios only,

1. while marking the referenced objects in the tenured generation space.
2. if there is a change in heap memory in parallel while doing the garbage collection.

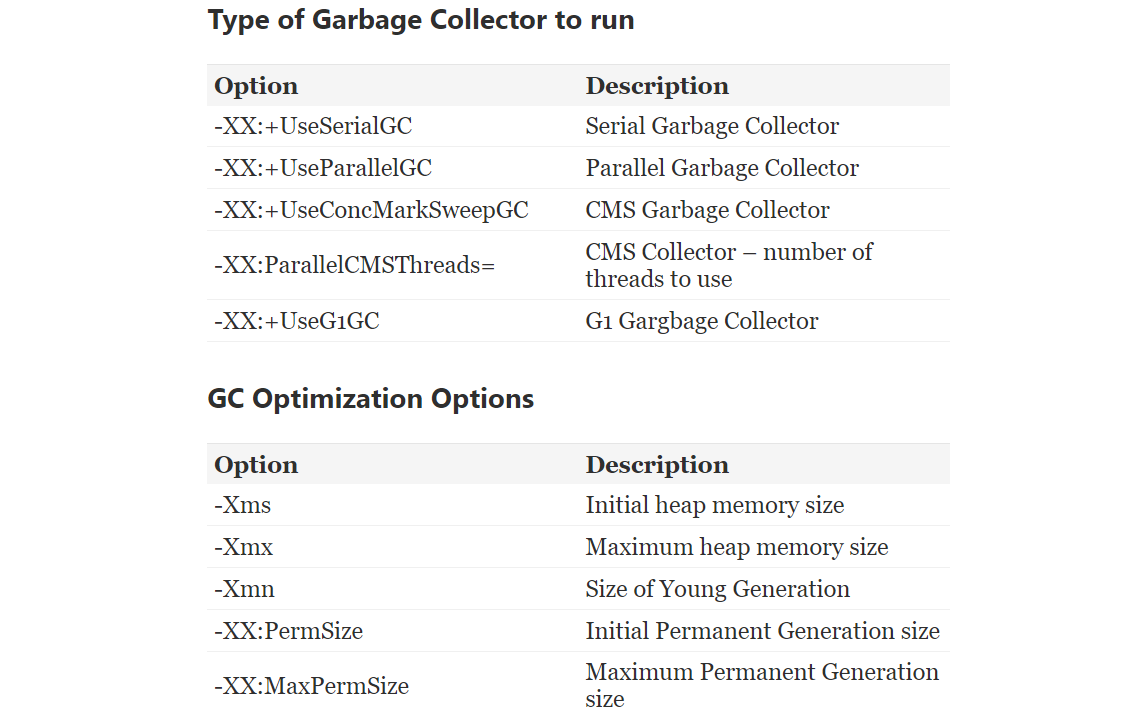
In comparison with parallel garbage collector, CMS collector uses more CPU to ensure better application throughput. If we can allocate more CPU for better performance then CMS garbage collector is the preferred choice over the parallel collector.

**G1 garbage collector** is used for large heap memory areas. It separates the heap memory into regions and does collection within them in parallel. G1 also does compacts the free heap space on the go just after reclaiming the memory. But CMS garbage collector compacts the memory on stop the world (STW) situations. G1 collector prioritizes the region based on most garbage first.

**Java 8 Improvement:**

Turn on the -XX:+UseStringDeduplication JVM argument while using G1 garbage collector. This optimizes the heap memory by removing duplicate String values to a single char[] array. This option is introduced in Java 8 u 20.

*Given all the above four types of Java garbage collectors, which one to use depends on the application scenario, hardware available and the throughput requirements.*



java -Xmx12m -Xms3m -Xmn1m -XX:PermSize=20m -XX:MaxPermSize=20m -XX:+UseSerialGC -jar java-application.jar

**17 - What is immutable class and how can we achieve immutable class?**

* Immutable class in java means that once an object is created, we cannot change its content.
* In Java, all the wrapper classes (like Integer, Boolean, Byte, Short) and String class is immutable.

Rules:

* The class must be declared as **final** so that child classes can’t be created.
* Data members in the class must be declared **private** so that direct access is not allowed.
* Data members in the class must be declared as **final** so that we can’t change the value of it after object creation.
* A **parameterized** **constructor** should initialize all the fields performing a deep copy so that data members can’t be modified with an object reference.
* **Deep Copy (Deep Clone)** of objects should be performed in the getter methods to return a copy rather than returning the actual object reference)

public final class EmployeeImmutable {  
 private final int id;  
 private Address address;  
  
 public EmployeeImmutable(int id, Address address) {  
 this.id = id;  
 this.address = address;  
 }  
  
 public int getId() {  
 return id;  
 }  
 public Address getAddress() throws CloneNotSupportedException {  
 return (Address) address.clone();  
 }  
}

public class Address implements Cloneable {  
  
 private String addressType;  
 private String address;  
 private String city;  
  
 public Address(String addressType, String address, String city) {  
 super();  
 this.addressType = addressType;  
 this.address = address;  
 this.city = city;  
 }  
  
 public String getAddressType() {  
 return addressType;  
 }  
 public void setAddressType(String addressType) {  
 this.addressType = addressType;  
 }  
 public String getAddress() {  
 return address;  
 }  
 public void setAddress(String address) {  
 this.address = address;  
 }  
 public String getCity() {  
 return city;  
 }  
 public void setCity(String city) {  
 this.city = city;  
 }  
 public Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
  
 @Override  
 public String toString() {  
 return "Address Type - " + addressType + ", address - " + address + ", city - " + city;  
 }  
}

**18. Cloning in Java**

Clone is nothing but the process of copying one object to produce the exact object, which is not guaranteed.

So, we can define Cloning as “**create a copy of object** “. I think now we are somehow clear about cloning but there is more to it depending on how we are doing this copy, we can divide cloning into two types.

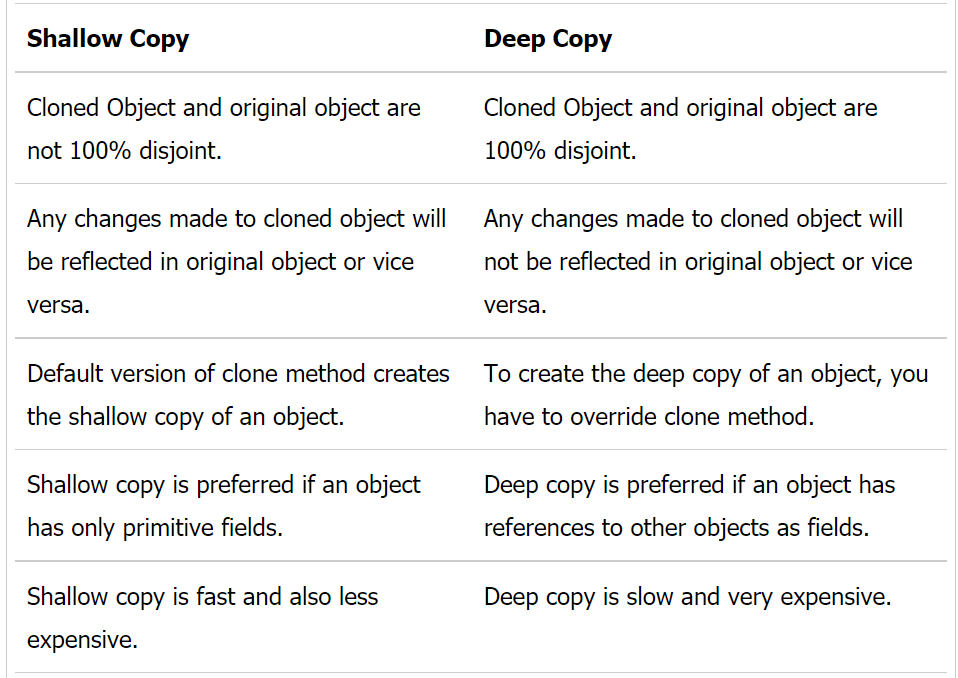
* Shallow Copy
* Deep Copy

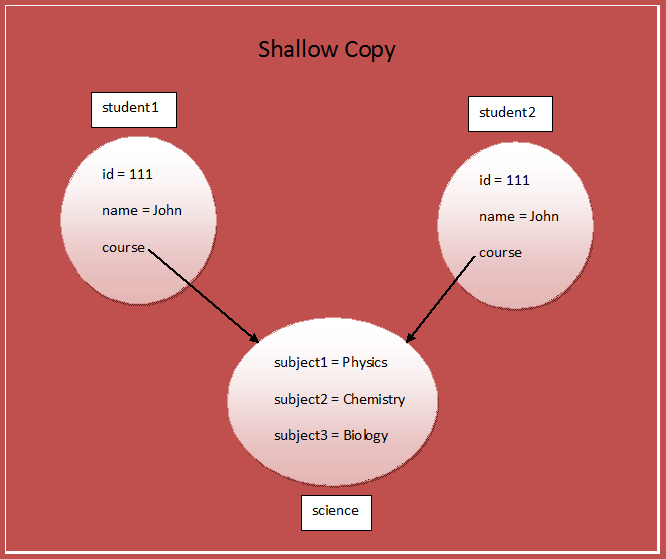
In Java, everything is achieved through class, object, and interface. By default, no Java class support cloning but Java provide one interface called Cloneable, which is a marker interface and by implementing this interface we can make the duplicate copy of our object by calling **clone()** method of  **java.lang.Object** class.

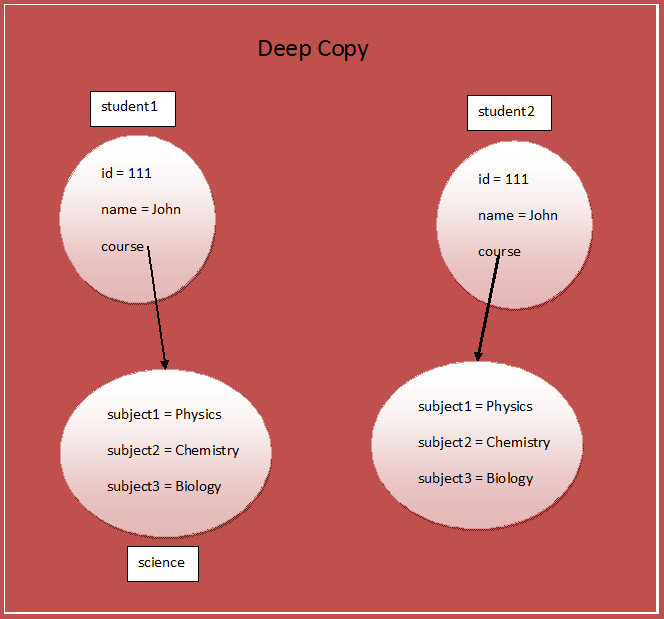
This Method is protected inside the object class and Cloneable interface is a marker interface and this method also throws CloneNotSupportedException, if we have not implemented this interface and try to call clone() method of Object class.

**By default any clone() method gives the shallow copy of the object** i.e. if we invoke super.clone() then it’s a shallow copy but if we want to **deep copy we have to override the clone() method and make it public** and give own definition of making copy of object. Now we let’s see what is shallow and deep copy of object in Java programming language.

**Difference between Shallow and Deep Copy in Java**







class Course {  
 String subject1;  
 String subject2;  
 String subject3;  
  
 public Course(String sub1, String sub2, String sub3) {  
 this.subject1 = sub1;  
 this.subject2 = sub2;  
 this.subject3 = sub3;  
 }  
}

class Student implements Cloneable {  
 int id;  
 String name;  
 Course course;  
  
 public Student(int id, String name, Course course) {  
 this.id = id;  
 this.name = name;  
 this.course = course;  
 }  
 //Default version of clone() method. It creates shallow copy of an object.  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}

public class ShallowCopyInJava {  
 public static void main(String[] args) {  
 Course science = new Course("Physics", "Chemistry", "Biology");  
 Student student1 = new Student(111, "John", science);  
 Student student2 = null;  
  
 try {  
 //Creating a clone of student1 and assigning it to student2  
 student2 = (Student) student1.clone();  
 } catch (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
  
 //Printing the subject3 of 'student1'  
 System.*out*.println(student1.course.subject3); //Output : Biology  
  
 //Changing the subject3 of 'student2'  
 student2.course.subject3 = "Maths";  
  
 //This change will be reflected in original student 'student1'  
 System.*out*.println(student1.course.subject3); //Output : Maths  
 }  
}

--

class CourseDeep implements Cloneable {  
 String subject1;  
 String subject2;  
 String subject3;  
 public CourseDeep(String sub1, String sub2, String sub3) {  
 this.subject1 = sub1;  
 this.subject2 = sub2;  
 this.subject3 = sub3;  
 }  
 protected Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
}

class StudentDeep implements Cloneable {  
 int id;  
 String name;  
 Course course;  
  
 public StudentDeep(int id, String name, CourseDeep course) {  
 this.id = id;  
 this.name = name;  
 this.course = course;  
 }  
  
 //Overriding clone() method to create a deep copy of an object.  
 protected Object clone() throws CloneNotSupportedException {  
 StudentDeep student = (StudentDeep) super.clone();  
 student.course = (CourseDeep) course.clone();  
 return student;  
 }  
}

public class DeepCopyInJava {  
 public static void main(String[] args) {  
 CourseDeep science = new CourseDeep("Physics", "Chemistry", "Biology");  
 StudentDeep student1 = new StudentDeep(111, "John", science);  
 StudentDeep student2 = null;  
 try {  
 student2 = (StudentDeep) student1.clone();  
 } catch (CloneNotSupportedException e) {  
 e.printStackTrace();  
 }  
 //Printing the subject3 of 'student1'  
 System.*out*.println(student1.course.subject3); //Output : Biology  
 //Changing the subject3 of 'student2'  
 student2.course.subject3 = "Maths";  
 //This change will not be reflected in original student 'student1'  
 System.*out*.println(student1.course.subject3); //Output : Biology  
 }  
}

**19- Create a collection that has limitation on max entries.**

public class LimitedHashSet<E> extends HashSet<E> {  
 private static final long *serialVersionUID* = -23456691722L;  
 private final int limit;  
  
 public LimitedHashSet(int limit) {  
 this.limit = limit;  
 }  
  
 @Override  
 public boolean add(E object) {  
 if (this.size() > limit) {  
 return false;  
 }  
 return super.add(object);  
 }  
}