**JAVA Interview Important Questions**

# What are OOPS concept in java?

Answer: Java is based on Object Oriented Programming Concepts, following are some of the OOPS concepts implemented in java programming.

* **Abstraction**: Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

* **Encapsulation**: Encapsulation is one of the four fundamental OOP concepts. The other three are inheritance, polymorphism, and abstraction.

Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit. In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class. Therefore, it is also known as **data hiding**.

To achieve encapsulation in Java −

Declare the variables of a class as private.

Provide public setter and getter methods to modify and view the variables values.

* **Polymorphism**: Polymorphism means "many forms". In simple words, we can define polymorphism as the ability of a data to be displayed in more than one form.

In Java polymorphism is mainly divided into two types:

* **Compile time Polymorphism**: It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading. But Java doesn’t support the Operator Overloading. When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.
* **Runtime Polymorphism:** It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding which occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden**.**
* **Inheritance**
* **Association, Aggregation, Composition**:



Association is relation between two separate classes which establishes through their Objects. Association can be one-to-one, one-to-many, many-to-one, many-to-many.

In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. Composition and Aggregation are the two forms of association.

**Aggregation is a special form of Association** where:

* It represents Has-A relationship.
* It is a unidirectional association i.e. a one way relationship. For example, department can have students but vice versa is not possible and thus unidirectional in nature.
* In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity.
* When do we use Aggregation ?? Code reuse is best achieved by aggregation.
* For Example: Student and Department.

**Composition is a restricted form of Aggregation** in which two entities are highly dependent on each other.

* It represents part-of relationship.
* In composition, both the entities are dependent on each other.
* When there is a composition between two entities, the composed object cannot exist without the other entity.
* For Example: Room and Home

**Aggregation vs Composition**

* **Dependency**: Aggregation implies a relationship where the child can exist independently of the parent. For example, Bank and Employee, delete the Bank and the Employee still exist. whereas Composition implies a relationship where the child cannot exist independent of the parent. Example: Human and heart, heart don’t exist separate to a Human
* **Type of Relationship**: Aggregation relation is “has-a” and composition is “part-of” relation.
* **Type of association**: Composition is a strong Association whereas Aggregation is a weak Association.

# What are various Thread execution prevention methods?

Answer: yield(),join() and sleep()

# Difference Between yield(), join() and sleep()?

Answer:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **yield()** | **join()** | **sleep()** |
| **purpose** | If a thread wants to pass its execution to give chance to remaining threads of same priority, then we should go for yield() | If a thread wants to wait until completing of some other thread, then we should go for join() | If a thread does not want to perform any operation for a particular amount of time, then it goes for sleep() |
| **Is it overloaded?** | NO | YES | YES |
| **Is it final?** | NO | YES | NO |
| **Is it throws?** | NO | YES | YES |
| **Is it native?** | YES | NO | sleep(long ms)->native & sleep (long ms, int ns)-> non native |
| **Is it static?** | YES | NO | YES |

# What is the various Thread Communication Methods?

Answer: wait(), notify() and notifyAll()

# Difference Between wait() and sleep()?

Answer:

* The fundamental difference is that wait() is from Object and sleep() is a static method of Thread.
* The major difference is that wait() releases the lock while sleep() doesn’t release any lock while waiting.
* wait() is used for inter-thread communication while sleep() is used to introduce a pause on execution, generally.
* wait() should be called from inside synchronise or else we get an IllegalMonitorStateException, while sleep() can be called anywhere.
* To start a thread again from wait(), you have to call notify() or notifyAll(). As for sleep(), the thread gets started after a specified time interval.

**Similarities:**

* Both make the current thread go into the Not Runnable state.
* Both are native methods.

**Other Important Points**:

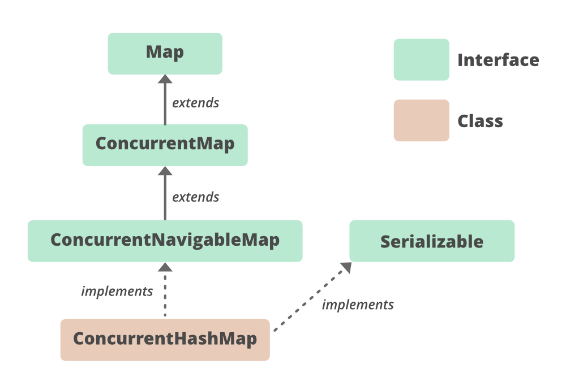
* **Method called on:**
* wait() – Call on an object; current thread must synchronize on the lock object.
* sleep() – Call on a Thread; always currently executing thread.
* **Synchronized:**
* wait() – when synchronized multiple threads access same Object one by one.
* sleep() – when synchronized multiple threads wait for sleep over of sleeping thread.
* **Lock duration:**
* wait() – release the lock for other objects to have chance to execute.
* sleep() – keep lock for at least t times if timeout specified or somebody interrupt.
* **wake up condition:**
* wait() – until call notify(), notifyAll() from object
* sleep() – until at least time expire or call interrupt().
* **Usage**
* sleep() – for time-synchronization
* wait() – for multi-thread-synchronization.

# What is concurrent HashMap?

Answer: The ConcurrentHashMap class is introduced in JDK 1.5 belongs to java.util.concurrent package, which implements ConcurrentMap as well as Serializable interface also. ConcurrentHashMap is an enhancement of HashMap as we know that while dealing with Threads in our application HashMap is not a good choice because performance-wise HashMap is not up to the mark.

**Key points of ConcurrentHashMap**:

* The underlined data structure for ConcurrentHashMap is Hashtable.
* ConcurrentHashMap class is thread-safe i.e. multiple threads can operate on a single object without any complications.
* At a time, any number of threads are applicable for a read operation without locking the ConcurrentHashMap object which is not there in HashMap.
* In ConcurrentHashMap, the Object is divided into a number of segments according to the concurrency level.
* The default concurrency-level of ConcurrentHashMap is 16.
* In ConcurrentHashMap, at a time any number of threads can perform retrieval operation but for updation in the object, the thread must lock the particular segment in which the thread wants to operate. This type of locking mechanism is known as Segment locking or bucket locking. Hence at a time, 16 update operations can be performed by threads.
* Inserting null objects is not possible in ConcurrentHashMap as key or value.



**Constructors of ConcurrentHashMap:**

* **Concurrency-Level**: It is the number of threads concurrently updating the map. The implementation performs internal sizing to try to accommodate this many threads.
* **Load-Factor**: It’s a threshold, used to control resizing.
* **Initial Capacity**: Accommodation of a certain number of elements initially provided by the implementation. if the capacity of this map is 10. It means that it can store 10 entries.

**Difference between HashMap and ConcurrentHashMap:**

HashMap is the Class which is under Traditional Collection and ConcurrentHashMap is a Class which is under Concurrent Collections, apart from this there are various differences between them which are:

* HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
* HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
* While one thread is Iterating the HashMap object, if other thread tries to add/modify the contents of Object then we will get Run-time exception saying ConcurrentModificationException. Whereas In ConcurrentHashMap we won’t get any exception while performing any modification at the time of Iteration.
* In HashMap, null values are allowed for key and values, whereas in ConcurrentHashMap null value is not allowed for key and value, otherwise we will get Run-time exception saying NullPointerException.
* HashMap is introduced in JDK 1.2 whereas ConcurrentHashMap is introduced by SUN Microsystem in JDK 1.5.
* If we try to modify the collection while iterating over it, we get ConcurrentModificationException. Java 1.5 introduced Concurrent classes in the java.util.concurrent package to overcome this scenario. ConcurrentHashMap is the Map implementation that allows us to modify the Map while iteration. The ConcurrentHashMap operations are thread safe. ConcurrentHashMap doesn’t allow null for keys and values.
* Iterator on Collection objects is fail-fast i.e. any modification in the structure or the number of entries in the collection object will trigger the exception.
* A ConcurrentHashMap is divided into number of segments, and the example which I am explaining here used default as 32 on initialization.
* A ConcurrentHashMap has internal final class called Segment so we can say that ConcurrentHashMap is internally divided in segments of size 32, so at max 32 threads can work at a time. It means each thread can work on each segment during high concurrency and atmost 32 threads can operate at max which simply maintains 32 locks to guard each bucket of the ConcurrentHashMap.

# Whether null insertion is possible in vector? How many null insertions are possible in vector?

Answer: yes, any number

# What are legacy classes and name any 3 legacy classes or interfaces in collection?

Answer: Early version of java did not include the Collections framework. It only defined several classes and interfaces that provide methods for storing objects. When Collections framework were added in J2SE 1.2, the original classes were reengineered to support the collection interface. These classes are also known as Legacy classes

**For Example**: Vector, Hashtable, Properties, Stack, Dictionary

There is only one legacy interface called **Enumeration**

# By default, every collection implements which interfaces?

Answer: Cloneable and Serializable

# Difference Between ArrayList Vs LinkedList?

Answer: 1) **Search**: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

**Reason**: ArrayList implements **RandomAccess** interface which maintains index-based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements doubly linked list which requires the traversal through all the elements for searching an element.

2) **Deletion**: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

**Conclusion**: LinkedList element deletion is faster compared to ArrayList.

**Reason**: LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While in ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) **Inserts** **Performance**: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

4) **Memory** **Overhead**: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

# How sorting is done in Collection?

Answer: Collections.sort(), Comparator, Comparable, TreeSet and TreeMap.

# Explain Equals and hashCode contract in Java?

Answer:

* If two objects are equal by equals() method, then there hashcode must be same.
* If two objects are not equal by equals() method, then there hashcode could be same or different.

# Difference between Comparator and Comparable?

Answer:

|  |  |
| --- | --- |
| **Java.lang.Comparable** | **Java.util.Comparator** |
| Comparable provides a single sorting sequence. In other words, we can sort the collection on the basis of a single element such as id, name, and price. | The Comparator provides multiple sorting sequences. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. |
| The comparable interface has a method compareTo(Object a) to sort elements. | The comparator has a method compare(Object o1, Object O2) to sort elements. |
| Collections.sort(List) method can be used to sort the collection of Comparable type objects. | Collections.sort(List, Comparator) method can be used to sort the collection of Comparator type objects. |
| Comparable affects the original class, i.e., the actual class is modified. | Comparator doesn't affect the original class, i.e., the actual class is not modified. |
| When we make a collection element comparable(by having it implement Comparable), we get only one chance to implement the compareTo() method | Unlike Comparable, Comparator is external to the element type we are comparing. It’s a separate class. We create multiple separate classes (that implement Comparator) to compare by different members. |
| Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example, Roll Numbers of students. | Comparator interface sorting is done through a separate class |
| Logically, Comparable interface compares “this” reference with the object specified | Comparator in Java compares two different class objects provided. |

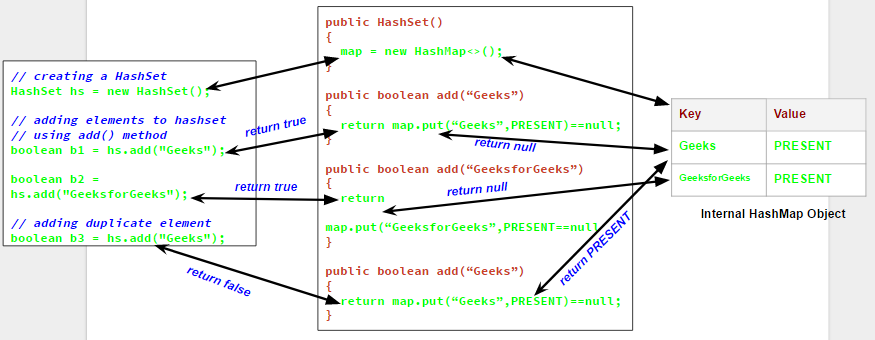
To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if sorting needs to be done on attributes of different objects, then use Comparator in Java.

# Explain the Internal working of HashSet?

Answer: A Set is a well-defined collection of distinct objects. Each member of a set is called an element of the set. So, in other words, we can say that a set will never contain duplicate elements.

// Dummy value to associate with an Object in the backing Map

**private static final Object PRESENT = new Object();**



Now as you can see that whenever we create a HashSet, it internally creates a HashMap and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as its key and constant Object called “PRESENT” as its value. So, we can say that a Set achieves uniqueness internally through HashMap.

* If the method map.put(key, value) returns null, then the method map.put(e, PRESENT)==null will return true internally, and the element added to the HashSet.
* If the method map.put(key, value) returns the old value of the key, then the method map.put(e, PRESENT)==null will return false internally, and the element will not add to the HashSet.

**Retrieving Object from the HashSet**

We use iterator() method to retrieve object from the HashSet. It is a method of java.util.HashSet class. It returns iterator for backup Map returned by map.keySet().iterator() method.

public Iterator<E> iterator()

{

return map.keySet().iterator();

}

# Explain the Internal working of HashMap?

Answer:

HashMap<K, V> is a part of Java’s collection since Java 1.2. This class is found in java.util package. It provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs, and you can access them by an index of another type (e.g. an Integer). One object is used as a key (index) to another object (value). If you try to insert the duplicate key, it will replace the element of the corresponding key.

HashMap is similar to the Hashtable, but it is unsynchronized. It allows to store the null keys as well, but there should be only one null key object and there can be any number of null values. This class makes no guarantees as to the order of the map. To use this class and its methods, you need to import java.util.HashMap package or its superclass.

HashMap in Java works on hashing principles. It is a data structure which allows us to store object and retrieve it in constant time O(1) provided we know the key. In hashing, hash functions are used to link key and value in HashMap. Objects are stored by calling put(key, value) method of HashMap and retrieved by calling get(key) method. When we call put method, the hashcode() method of the key object is called so that the hash function of the map can find a bucket location to store value object, which is actually an index of the internal array, known as the table.

HashMap like HashMap accept null while Hashtable doesn't, HashMap is not synchronized, HashMap is fast

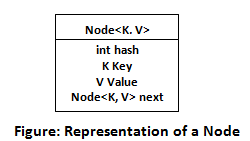
**What is Hashing**

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

**What is HashMap**

HashMap internally stores mapping in the form of Map.Entry object which contains both key and value object.

HashMap is a part of the Java collection framework. It uses a technique called Hashing. It implements the map interface. It stores the data in the pair of Key and Value. HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.

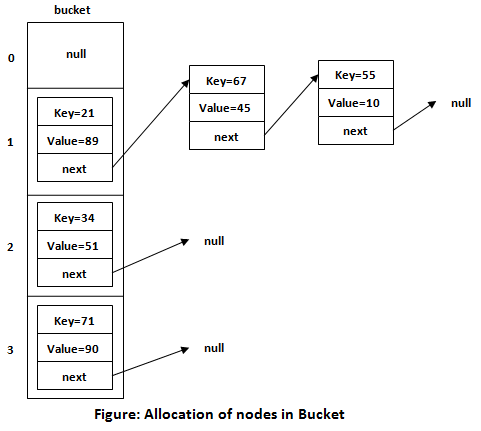


Before understanding the internal working of HashMap, you must be aware of hashCode() and equals() method.

**equals**(): It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.

**hashCode**(): This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.

**Buckets**: Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.



**Insert Key, Value pair in HashMap**

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

**Example**

In the following example, we want to insert three (Key, Value) pair in the HashMap.

HashMap<String, Integer> map = new HashMap<>();

map.put("Aman", 19);

map.put("Sunny", 29);

map.put("Ritesh", 39);

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

**Calculating Index**

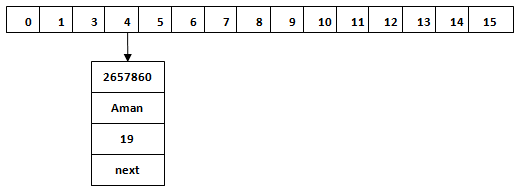
Index minimizes the size of the array. The Formula for calculating the index is:

**Index = hashcode(Key) & (n-1)**

Where n is the size of the array. Hence the index value for "Aman" is:

**Index = 2657860 & (16-1) = 4**

The value 4 is the computed index value where the Key and value will store in HashMap.

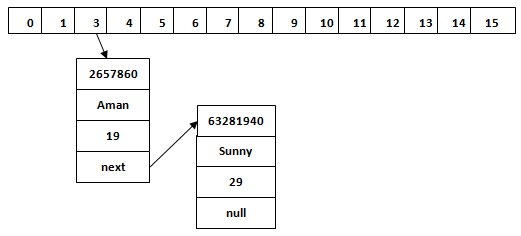


**Hash Collision**

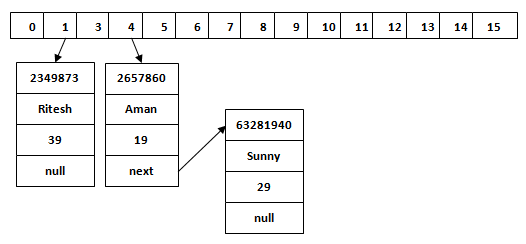
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

**Index=63281940 & (16-1) = 4**

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



**get() method in HashMap**

If we try to retrieve an object from this linked list, we need an extra check to search the correct value, this is done by equals() method. Since each node contains an entry, HashMap keeps comparing entry's key object with the passed key using equals() and when it returns true, Map returns the corresponding value.

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

**map.get(new Key("Aman"));**

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

Since searching inlined list is O(n) operation, in worst case hash collision reduces a map to linked list. This issue is recently addressed in Java 8 by replacing the linked list to the tree to search in O(logN) time.

# What happens On HashMap in Java if the size of the HashMap exceeds a given threshold defined by load factor? do you see any problem with resizing of HashMap in Java?

Answer: If the size of the HashMap exceeds a given threshold defined by load-factor e.g. if the load factor is .75 it will act to re-size the map once it filled 75%. Similar to other collection classes like ArrayList, Java HashMap re-size itself by creating a new bucket array of size twice of the previous size of HashMap and then start putting every old element into that new bucket array. This process is called rehashing because it also applies the hash function to find new bucket location.

What happen when multiple thread accessing the Java HashMap and potentially looking for race condition on HashMap in Java. Yes, there is potential race condition exists while resizing HashMap in Java, if two thread at the same time found that now HashMap needs resizing and they both try to resize. on the process of resizing of HashMap in Java, the element in the bucket which is stored in linked list get reversed in order during their migration to new bucket because Java HashMap doesn't append the new element at tail instead it append new element at the head to avoid tail traversing. If race condition happens then you will end up with an infinite loop.

# Explain Hashtable?

Answer: The Hashtable class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a Hashtable, the objects used as keys must implement the hashCode method and the equals method.

Features of Hashtable

* It is similar to HashMap but is synchronized.
* Hashtable stores key/value pair in hash table.
* In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
* The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.
* HashMap doesn’t provide any Enumeration, while Hashtable provides not fail-fast Enumeration.

# Difference between HashSet, LinkedHashset and TreeSet?

Answer: LinkedHashSet is the ordered version of HashSet. The only difference between HashSet and LinkedHashSet is that: LinkedHashSet maintains the insertion order. When we iterate through a HashSet, the order is unpredictable while it is predictable in case of LinkedHashSet.

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **HashSet** | **Linked Hash Set** | **Tree Set** |
| **Internal Working** | HashSet internally uses HashMap for storing objects | LinkedHashSet uses LinkedHashMap internally to store objects | TreeSet uses TreeMap internally to store objects |
| **When to Use** | If you don’t want to maintain insertion order but want store unique objects | If you want to maintain insertion order of elements, then you can use LinkedHashSet | If you want to sort the elements according to some Comparator, then use TreeSet |
| **Order** | HashSet does not maintain insertion order | LinkedHashSet maintains insertion order of objects | While TreeSet orders of the elements according to supplied Comparator. Default, it’s objects will be placed in their natural ascending order. |
| **Complexity of Operations** | HashSet gives O(1) complicity for insertion, removing and retrieving objects | LinkedHashSet gives insertion, removing and retrieving operations performance in order O(1). | While TreeSet gives performance of order O(log(n)) for insertion, removing and retrieving operations. |
| **Performance** | HashSet performance is better according to LinkedHashSet and TreeSet. | The performance of LinkedHashSet is slow to TreeSet. The performance LinkedHashSet is almost similar to HashSet but slower because, LinkedHashSet maintains LinkedList internally to maintain the insertion order of elements | TreeSet performance is better to LinkedHashSet excluding insertion and removal operations because, it has to sort the elements after each insertion and removal operations. |
| **Compare** | HashSet uses equals() and hashCode() methods to compare the objects | LinkedHashSet uses equals() and hashCode() methods to compare its objects | TreeSet uses compare() and compareTo() methods to compare the objects |
| **Null Elements** | HashSet allows only one null objects | LinkedHashSet allows only one null objects. | TreeSet not allow any null objects. If you insert null objects into TreeSet, it throws NullPointerException |

**Similarities Between HashSet, LinkedHashSet and TreeSet**:

* **Duplicates** : HashSet, LinkedHashSet and TreeSet are implements Set interface so they are not allowed to store duplicates objects.
* **Thread safe** : If we want to use HashSet, LinkedHashSet and TreeSet in multi-threading environment then first we make it externally synchronize because both LinkedHashSet and TreeSet are not thread safe.
* All three are **Cloneable and Serializable**.

# Why Should an Object Used as the Key in HashMap should be Immutable? OR Why String, Integer and other wrapper classes are considered good keys?

Answer: 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. Other wrapper class also shares similar property. Immutability is required, in order to prevent changes on fields used to calculate hashCode() because if key object returns different hashCode during insertion and retrieval than it won't be possible to get an object from HashMap.

Immutability is best as it offers other advantages as well like thread-safety, If you can keep your hashCode same by only making certain fields final, then you go for that as well. Since equals() and hashCode() method is used during retrieval of value object from HashMap, it's important that key object correctly override these methods and follow contact. If unequal object returns different hashcode than chances of collision will be less which subsequently improve the performance of HashMap.

# How to make a class immutable in java?

Answer:

* Make your class final, so that no other classes can extend it.
* Make all your fields final, so that they’re initialized only once inside the constructor and never modified afterward.
* Don’t expose setter methods.
* When exposing methods which modify the state of the class, you must always return a new instance of the class.
* If the class holds a mutable object:
* Inside the constructor, make sure to use a clone copy of the passed argument and never set your mutable field to the real instance passed through constructor, this is to prevent the clients who pass the object from modifying it afterwards.
* Make sure to always return a clone copy of the field and never return the real object instance.

**OR**

* Declare the class as final so it can’t be extended.
* Make all fields private so that direct access is not allowed.
* Don’t provide setter methods for variables.
* Make all mutable fields final so that its value can be assigned only once.
* Initialize all the fields via a constructor performing deep copy.
* Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

# How to make a class singleton in java?

Answer: Singleton design pattern is the solution proposed to return same instance every time restrict instantiation of a class more than once exactly one copy is available at any given point of time ensures only one instance is available in a Java Virtual Machine Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

There are two forms of singleton design pattern

1. Early Instantiation: creation of instance at load time.
2. Lazy Instantiation: creation of instance when required.

To create the singleton class, we need to have static member of class, private constructor and static factory method.

* **Static member**: It gets memory only once because of static, it contains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method**: This provides the global point of access to the Singleton object and returns the instance to the caller.

**early Instantiation of Singleton Pattern**:

class A{

private static A obj=new A();//Early, instance will be created at load time

private A(){}

public static A getA(){

return obj;

}

public void doSomething(){

//write your code

}

}

**lazy Instantiation of Singleton Pattern:**

class A{

private static A obj;

private A(){}

public static A getA(){

if (obj == null){

synchronized(Singleton.class){

if (obj == null){

obj = new Singleton();//instance will be created at request time

}

}

}

return obj;

}

public void doSomething(){

//write your code

}

}

**Significance of Classloader in Singleton Pattern:**

If singleton class is loaded by two classloaders, two instance of singleton class will be created, one for each classloader.

# Explain singleton pattern in multithreaded environment in java?

Answer:

**Let us understand in steps**:

* Thread-1 got the chance and it is put into execution
* It finds the INSTANCE to be null and therefore Thread-1 instantiates
* Concurrently if any other thread got chance and if it tries to execute, then there may be a possibility of new instance is getting created, although it is 50 % chance
* Because, new Thread-1 haven’t completed with creation of singleton INSTANCE and another thread at the same time finds singleton INSTANCE to be null and tries to create another one

To overcome this situation, we need to execute lazy instance creation inside synchronized block

Basic steps to create Singleton class using Lazy Initialization in a Multi-threaded environment

* **Step 1**: Just declare private static variable of the same class (beware don’t instantiate)
* **Step 2**: Provide private constructor to restrict instantiation from outside class
* **Step 3**: Provide public static getInstance() method and check
  + **Step 3.a**: If INSTANCE variable is null, then only instantiate
  + **Step 3.b**: Otherwise, return already instantiated INSTANCE variable
  + **Synchronized**: Put both above checks inside synchronized block
* **Step 4**: In addition to above detailed steps, also make INSTANCE variable as volatile. This will help getting latest updated copy every time, as it will read from main memory than in its own CPU-cache area
* **Note**: If your singleton INSTANCE is going to be executed in a single threaded environment, then there is no need of making INSTANCE variable as volatile

# Explain singleton pattern in multi JVM environment?

Answer: Singleton design pattern comes into picture when we have to control the number of object instances that can be created per Class per JVM. Singleton ensures only one instance of the class in a given JVM. one for each JVM.

# How do you prevent SQL Injection in Java Code?

Answer: You can use Prepared Statement to avoid SQL injection in Java code

# How does Concurrent HashMap achieve its Scalability?

Answer: The Concurrent HashMap class solves this problem by using a lock stripping technique, where the whole map is locked at different segments and only a particular segment is locked during the write operation, not the whole map. The concurrent HashMap also achieves its scalability by allowing lock-free reads as read is a thread-safe operation. See here for more advanced multi-threading and concurrency questions in Java.

# What is the output of the given Java code?

public class Test {

public static void main(String[] args) {  
 method(null);  
 }  
 public static void method(Object o) {  
 System.out.println("Object method");  
 }  
 public static void method(String s) {  
 System.out.println("String method");  
 }}

Answer: It will print “String method”. First of all, null is not an object in Java. But we know that we can assign null to any object reference type in Java. Java String is also an object of the class java.lang.String. Here, the Java compiler chooses to call the overloaded method with the most specific parameters. Which would be String because the String class is more specific than the Object class.

# What is the output of the given Java code?

public class Test{

public static void main(String[] args){

Integer num1 = 100;

Integer num2 = 100;

if(num1==num2){

System.out.println("num1 == num2");

}

else{

System.out.println("num1 != num2");

}} }

Answer: It will print “num1 == num2”. Whenever two different object references are compared using “==,” the value is always “false.” But here, because of the Integer caching, num1 and num2 are autoboxed. Thus num1==num2 returns “true”. Integer caching happens only for values between -128 and 127.

# Is Java “pass-by-reference” or “pass-by-value”?

Answer: Java is always “pass-by-value”. However, when we pass the value of an object, we pass the reference to it because the variables store the object reference, not the object itself. But this isn’t “pass-by-reference.” This could be confusing for beginners.

# How many String objects are created by the below code?

String s = new String("Hello World");  
Answer: Two String objects are created. When the new operator is used to create a String object, if the object does not exist in the Java String Pool, it will first be created in it, and then in the heap memory as well. You can simply learn all about Java Strings in my article below.

# Is it possible to override or overload a static method in Java?

Answer: It’s possible to overload static Java methods, but it’s not possible to override them. You can write another static method with the same signature in the subclass, but it’s not going to override the superclass method. It’s called method hiding in Java.

# What do the expression 1.0 / 0.0 will return? will it throw Exception? any compile time error?

Answer: This is another tricky question from Double class. Though Java developer knows about the double primitive type and Double class, while doing floating point arithmetic they don't pay enough attention to Double.INFINITY, NaN, and -0.0 and other rules that govern the arithmetic calculations involving them. The simple answer to this question is that it will not throw ArithmeticExcpetion and return Double.INFINITY. Also, note that the comparison x == Double.NaN always evaluates to false, even if x itself is a NaN. To test if x is a NaN, one should use the method call Double.isNaN(x) to check if given number is NaN or not. If you know SQL, this is very close to NULL there.

# Explain various Method Overriding rules in java?

Answer:

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the subclass cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.

# What are Method Overloading rules in Java?

Answer:

* Overloaded methods are differentiated based on the number and type of the parameters passed as an argument to the methods.
* You cannot define more than one method with the same name, Order and the type of the arguments. It would be compiler error.
* The compiler does not consider the return type while differentiating the overloaded method. But you cannot declare two methods with the same signature and different return type. It will throw a compile time error.
* If both methods have same parameter types, but different return type, then it is not possible.

**Method overloading can be done by changing**:

* The number of parameters in two methods.
* The data types of the parameters of methods.
* The Order of the parameters of methods.

# What is abstract class in java?

Answer:

# What are interfaces in java?

Answer:

**There are mainly three reasons to use interface.**

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

**Interfaces have the following properties**

* An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
* Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
* Methods in an interface are implicitly public.
* If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
* The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

**New features added in interfaces in JDK 8**

* Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.

Suppose we need to add a new function in an existing interface. Obviously, the old code will not work as the classes have not implemented those new functions. So, with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

* Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

**New features added in interfaces in JDK 9**

* Private methods:
* Allows default methods to share common code to avoid duplicate code (redundancy)
* Improves readability of code.
* Private interface method cannot be abstract.
* Private method can be used only inside interface.
* Private static method can be used inside other static and non-static interface methods.
* Private non-static methods cannot be used inside private static methods
* Private Static methods:
* Since java 8 we can have static methods in interfaces along with default methods. We cannot share the common code of static methods using the non-static private method, we must have to use the private static method to do that.

# Difference between abstract class and interface in java?

Answer:

|  |  |
| --- | --- |
| **Abstract Class** | **Interface** |
|  |  |
|  |  |
| An abstract class may contain non-final variables. | Variables declared in a Java interface are by default final |
|  |  |
|  |  |

# What is difference between CyclicBarrier and CountDownLatch in Java.

Answer:

# What is the difference between StringBuffer and StringBuilder in Java?

Answer: StringBuilder in Java was introduced in JDK 1.5 and the only difference between both of them is that StringBuffer methods e.g. length(), capacity() or append() are synchronized while corresponding methods in StringBuilder are not synchronized.

# What are the important features of Java 11 release?

# Name some important features of Java 10 release?

# Name some important features of Java 9 release?

# What are the important features of Java 8 release?

# Name some OOPS Concepts in Java?

# What do you mean by platform independence of Java?

# What is JVM and is it platform independent?

# What is the difference between JDK and JVM?

# What is the difference between JVM and JRE?

# Which class is the superclass of all classes?

# Why Java doesn’t support multiple inheritance?

# Why Java is not pure Object-Oriented language?

# What is difference between path and classpath variables?

# What is the importance of main method in Java?

# What is overloading and overriding in java?

# Can we overload main method?

# Can we have multiple public classes in a java source file?

# What is Java Package and which package is imported by default?

# What are access modifiers?

# What is final keyword?

# What is static keyword?

# What is finally and finalize in java?

# Can we declare a class as static?

# What is static import?

# What is try-with-resources in java?

# What is multi-catch block in java?

# What is static block?

# Can an interface implement or extend another interface?

# What is Marker interface?

# What are Wrapper classes?

# What is Enum in Java?

# What is Java Annotations?

# What is Java Reflection API? Why it’s so important to have?

# What is composition in java?

# What is the benefit of Composition over Inheritance?

# How to sort a collection of custom Objects in Java?

# What is inner class in java?

# What is anonymous inner class?

# What is Classloader in Java?

# What are different types of classloaders?

# What is ternary operator in java?

# What does super keyword do?

# What is break and continue statement?

# What is this keyword?

# What is default constructor?

# Can we have try without catch block?

# What is Garbage Collection?

# What is Serialization and Deserialization?

# How to run a JAR file through command prompt?

# What is the use of System class?

# What is instanceof keyword?

# Can we use String with switch case?

# Java is Pass by Value or Pass by Reference?

# What is difference between Heap and Stack Memory?

# Java Compiler is stored in JDK, JRE or JVM?

# What will be the output of following programs?

<https://stackoverflow.com/questions/4927227/why-doesnt-java-util-list-implement-serializable/4927292>

The interface body can contain [abstract methods](https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html), [default methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html), and [static methods](https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html#static). An abstract method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation). Default methods are defined with the default modifier, and static methods with the static keyword. All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.