1. OOPS Concepts:
   1. A.P.I.E :: abstraction, polymorphism, inheritance and encapsulation
   2. SOLID Design Principle
      1. [The Single Responsibility Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=2&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Fsrp.pdf&ei=1C4HSu3uEODJtgfs8uWeBw&usg=AFQjCNHQQ1Aw-2yCciEbERpJn3VdHBCmQw&sig2=Pf7Z4GNwRnAiaar_yCk_dg): A class (or method) should have only one reason to change.
      2. [The Open Closed Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Focp.pdf&ei=9i4HStWBH5XhtgfSltWNBw&usg=AFQjCNGzBuNcaA1IXUx0tijZoTW7rlzcRQ&sig2=K-zyAaPRTp0CG28mtuO9jg): A class (or method) should be open for extension and closed for modification.
      3. [The Liskov Substitution Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Flsp.pdf&ei=By8HSoDJJd6ptger2uSdBw&usg=AFQjCNFnNI0DmzofjWDQEGILAT-W1L8Mtw&sig2=ozd86zYF9ZMqLfBS2tQJdg): Subtypes must be substitutable for their base types.
      4. [The Interface Segregation Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Fisp.pdf&ei=Gy8HSpH-CqK_twfv6OCXBw&usg=AFQjCNHuKx3fPObvzYaQDZ2etcZHYtg57g&sig2=aWFqTuyl_iCAsGk8SnwljQ): Clients should not be forced to depend upon methods that they do not use. Interfaces should belong to clients.
      5. [The Dependency Inversion Principle](http://www.google.co.za/url?sa=t&source=web&ct=res&cd=1&url=http%3A%2F%2Fwww.objectmentor.com%2Fresources%2Farticles%2Fdip.pdf&ei=Mi8HSsbyLdKJtgemroCfBw&usg=AFQjCNEpBGziZw6APHj0rMG9pp1LJt9FHA&sig2=y4EFfBIw-cC6hocfVEeTJg): Abstractions should not depend on details. Details should depend on abstractions.

public Windows98Machine(Keyboard keyboard, Monitor monitor) {

this.keyboard = keyboard;

this.monitor = monitor;

}

* 1. Class design –SOLID principles

1. Core Java
   1. Abstract Class
   2. Inner class / static inner class/ Instance/ Anonymous --
   3. Immutable class
   4. Interface
   5. Marker Interface
   6. Exception handling
   7. Garbage Collection
   8. Class loader/ static and dynamic loading

Dyn loading::

try {

InputStreamReader in = new InputStreamReader(System.in);

BufferedReader reader = new BufferedReader(in);

System.out.println("Enter Class Name: ");

String whatClass = reader.readLine();

Class exampleClass = Class.forName(whatClass);

Object ob = exampleClass.newInstance();

} catch (ClassNotFoundException e) {

e.printStackTrace();

} catch (Exception e) {

e.printStackTrace();

}

* 1. Comparable / Comparator Interface
  2. String pool / String Buffer/ String Builder
  3. Constructor chaining / in case of abstract class/ interface
  4. This and super keywords
  5. Serialization
  6. Iterator / List Iterator
  7. Rules of overloading and overriding
  8. Reflection

For example, say you have an object of an unknown type in Java, and you would like to call a 'doSomething' method on it if one exists. Java's static typing system isn't really designed to support this unless the object conforms to a known interface, but using reflection, your code can look at the object and find out if it has a method called 'doSomething' and then call it if you want to.

So, to give you a code example of this in Java (imagine the object in question is foo) :

Method method = foo.getClass().getMethod("doSomething", null);

method.invoke(foo, null);

* 1. Exception Hierarchy and finally Method, Concurrent Modification Exception
  2. Pass by value / pass by reference
  3. Locking(class level vs instance level)

Object Level Locks − It can be used when you want non-static method or non-static block of the code should be accessed by only one thread.

Class Level locks − It can be used when we want to prevent multiple threads to enter the synchronized block in any of all available instances on runtime. It should always be used to make static data thread safe

* 1. Synchronization
  2. Locking mechanisms
  3. Serializable & Externalizable and cloanable Interfaces
  4. Finalize/clone method of object class

1. Collection Framework(internals)
   1. Array / Array List
   2. Linked List
   3. Vector (rarely asked)
   4. Hash Map
   5. hash Table
   6. Linked Hash Map
   7. Tree Map
   8. Sorted Map
   9. WeakHashMap

package com.example;

import java.util.HashMap;

import java.util.Map;

import java.util.WeakHashMap;

public class Example {

public static void main(String[] args) {

Key k1 = new Key("Hello");

Key k2 = new Key("World");

Key k3 = new Key("Java");

Key k4 = new Key("Programming");

Map<Key, String> hm = new HashMap<Key, String>();

hm.put(k1, "Hello");

hm.put(k2, "World");

hm.put(k3, "Java");

hm.put(k4, "Programming");

k1 = null;

k2 = null;

k3 = null;

k4 = null;

System.gc();

System.out.println("Hash Map :" + hm);

System.out.println("Same thing with weakHash Map - ");

k1 = new Key("Hello");

k2 = new Key("World");

k3 = new Key("Java");

k4 = new Key("Programming");

Map<Key, String> wm = new WeakHashMap<Key, String>();

wm.put(k1, "Hello");

wm.put(k2, "World");

wm.put(k3, "Java");

wm.put(k4, "Programming");

k1 = null;

k2 = null;

k3 = null;

k4 = null;

System.gc();

System.out.println("Weak Hash Map :" + wm);

}

}

class Key {

private String key;

public Key(String key) {

this.key = key;

}

@Override

public boolean equals(Object obj) {

return this.key.equals(obj);

}

@Override

public int hashCode() {

return key.hashCode();

}

@Override

public String toString() {

return key;

}

@Override

public void finalize() {

System.out.println("Finalize method is called");

}

}

**o/p**::

Hash Map :{Java=Java, Hello=Hello, Programming=Programming, World=World}

Same thing with weakHash Map -

Finalize method is called

Finalize method is called

Finalize method is called

Finalize method is called

Weak Hash Map :{}

* 1. All kinds of Sets/ Hash Set/Tree Set
  2. LinkedHashSet
  3. Stack / Queue/ Priority Queue/ Blocking Queue
  4. Condition Interface
  5. Fail safe and fail fast iterator
  6. CopyOnWriteArrayList

CopyOnWriteArrayList is a concurrent Collection class introduced in Java 5 Concurrency API along with its popular cousin ConcurrentHashMap in Java.

CopyOnWriteArrayList implements List interface like ArrayList, Vector and LinkedList but its a thread-safe collection and it achieves its thread-safety in a slightly different way than Vector or other thread-safe collection class.

As name suggest CopyOnWriteArrayList creates copy of underlying ArrayList with every mutation operation e.g. add or set. Normally CopyOnWriteArrayList is very expensive because it involves costly Array copy with every write operation but its very efficient if you have a List where Iteration outnumber mutation e.g. you mostly need to iterate the ArrayList and don't modify it too often.

Iterator of CopyOnWriteArrayList is fail-safe and doesn't throw ConcurrentModificationException even if underlying CopyOnWriteArrayList is modified once Iteration begins because Iterator is operating on separate copy of ArrayList. Consequently all the updates made on CopyOnWriteArrayList is not available to Iterator.

To get the most updated version do a new read like list.iterator();

That being said, updating this collection alot will kill performance. If you tried to sort a CopyOnWriteArrayList you'll see the list throws an UnsupportedOperationException (the sort invokes set on the collection N times). You should only use this read when you are doing upwards of 90+% reads.

* 1. ConcurrentSkipListMap

These two classes vary in a few ways.

ConcurrentHashMap does not guarantee\* the runtime of its operations as part of its contract. It also allows tuning for certain load factors (roughly, the number of threads concurrently modifying it).

ConcurrentSkipListMap, on the other hand, guarantees average O(log(n)) performance on a wide variety of operations. It also does not support tuning for concurrency's sake. ConcurrentSkipListMap also has a number of operations that ConcurrentHashMap doesn't: ceilingEntry/Key, floorEntry/Key, etc. It also maintains a sort order, which would otherwise have to be calculated (at notable expense) if you were using a ConcurrentHashMap.

Basically, different implementations are provided for different use cases. If you need quick single key/value pair addition and quick single key lookup, use the HashMap. If you need faster in-order traversal, and can afford the extra cost for insertion, use the SkipListMap.

* 1. ConcurrentHashMap

The point is to provide an implementation of HashMap that is threadsafe. Multiple threads can read from and write to it without the chance of receiving out-of-date or corrupted data. ConcurrentHashMap provides its own synchronization, so you do not have to synchronize accesses to it explicitly.

Another feature of ConcurrentHashMap is that it provides the putIfAbsent method, which will atomically add a mapping if the specified key does not exist. Consider the following code:

ConcurrentHashMap<String, Integer> myMap = new ConcurrentHashMap<String, Integer>();

// some stuff

if (!myMap.contains("key")) {

myMap.put("key", 3);

}

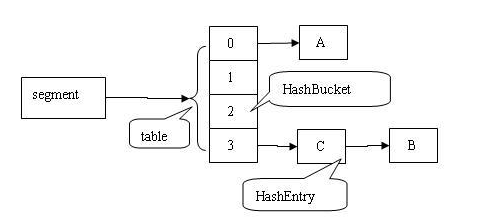
This code is not threadsafe, because another thread could add a mapping for "key" between the call to contains and the call to put. The correct implementation would be:

myMap.putIfAbsent("key", 3);

ConcurrentHashMap allow concurrent access to the map. HashTables too offers synchronized access to map, but your entire map is locked to perform any operation.

The logic behind ConcurrentHashMap is that your entire table is not getting locked, but only the portion[segments]. Each segments manages its own HashTable. Locking is applied only for updates. In case of of retrievals, it allows full concurrency.

Let's take four threads are concurrently working on a map whose capacity is 32, the table is partitioned into four segments where each segments manages a hash table of capacity. The collection maintains a list of 16 segments by default, each of which is used to guard (or lock on) a single bucket of the map.



This effectively means that 16 threads can modify the collection at a single time. This level of concurrency can be increased using the optional concurrencyLevel constructor argument.

public ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel)

As the other answer stated, the ConcurrentHashMap offers new method putIfAbsent() which is similar to put except the value will not be overridden if the key exists.

Private static Map<String,String> aMap =new ConcurrentHashMap<String,String>();

if(!aMap.contains("key"))

aMap.put("key","value");

The new method is also faster as it avoids double traversing as above. contains method has to locate the segment and iterate the table to find the key and again the method put has to traverse the bucket and put the key.

* 1. Collections.unmodifiableCollection()

1. Multithreading and Concurrency
   1. Thread lifecycle and basics
   2. Volatile

Volatile --> Guarantees visibility and NOT atomicity

Synchronization (Locking) --> Guarantees visibility and atomicity (if done properly)

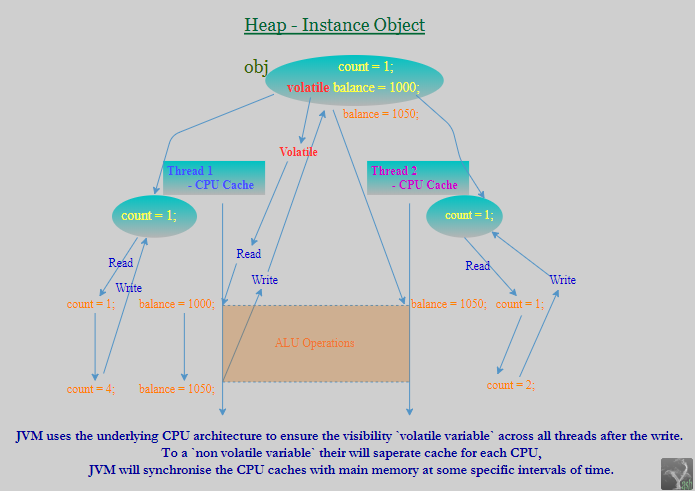
Volatile is not a substitute for synchronization

Use volatile only when you are updating the reference and not performing some other operations on it.

 Volatile Keyword is applicable to variables. volatile keyword in Java guarantees that value of the volatile variable will always be read from main memory and not from Thread's local cache.

Access\_Modifier volatile DataType Variable\_Name;

Volatile Field: An indication to the VM that multiple threads may try to access/update the field's value at the same time. To a special kind of instance variables which has to shared among all the threads with Modified value. Similar to Static(Class) variable, Only one copy of volatile value is cached in main memory, So that before doing any ALU Operations each thread has to read the updated value from Main memory after ALU operation it has to write to main memory direclty. (A write to a volatile variable v synchronizes-with all subsequent reads of v by any thread) ***This means that changes to a volatile variable are always visible to other threads.***



Here to a nonvoltaile variable if Thread t1 changes the value in t1's cache, Thread t2 can't access the changed value untill t1 writes, t2 read from main memory for the most recent modified value, which may lead to Data-Inconsistancy.

[**volatile cannot be cached**](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.5-200-A.1) - [assembler](http://hg.openjdk.java.net/icedtea/jdk7/hotspot/file/d9f0ed25f7ed/src/share/vm/opto/memnode.cpp#l2862)

+--------------+--------+-------------------------------------+

| Flag Name | Value | Interpretation |

+--------------+--------+-------------------------------------+

| ACC\_VOLATILE | 0x0040 | Declared volatile; cannot be cached.|

+--------------+--------+-------------------------------------+

|ACC\_TRANSIENT | 0x0080 | Declared transient; not written or |

| | | read by a persistent object manager.|

+--------------+--------+-------------------------------------+

[Shared Variables](https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4.1): Memory that can be shared between threads is called shared memory or heap memory. All instance fields, static fields, and array elements are stored in heap memory.

[Synchronization](https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-3.html#jvms-3.14): synchronized is applicable to methods, blocks. allows to execute only 1-thread at a time on object. If t1 takes control, then remaining threads has to wait untill it release the control.

Example:

public class VolatileTest implements Runnable {

private static final int MegaBytes = 10241024;

private static final Object counterLock = new Object();

private static int counter = 0;

private static volatile int counter1 = 0;

private volatile int counter2 = 0;

private int counter3 = 0;

@Override

public void run() {

for (int i = 0; i < 5; i++) {

concurrentMethodWrong();

}

}

void addInstanceVolatile() {

synchronized (counterLock) {

counter2 = counter2 + 1;

System.out.println( Thread.currentThread().getName() +"\t\t « InstanceVolatile :: "+ counter2);

}

}

public void concurrentMethodWrong() {

counter = counter + 1;

System.out.println( Thread.currentThread().getName() +" « Static :: "+ counter);

sleepThread( 1/4 );

counter1 = counter1 + 1;

System.out.println( Thread.currentThread().getName() +"\t « StaticVolatile :: "+ counter1);

sleepThread( 1/4 );

addInstanceVolatile();

sleepThread( 1/4 );

counter3 = counter3 + 1;

sleepThread( 1/4 );

System.out.println( Thread.currentThread().getName() +"\t\t\t\t\t « Instance :: "+ counter3);

}

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

Runtime runtime = Runtime.getRuntime();

int availableProcessors = runtime.availableProcessors();

System.out.println("availableProcessors :: "+availableProcessors);

System.out.println("MAX JVM will attempt to use : "+ runtime.maxMemory() / MegaBytes );

System.out.println("JVM totalMemory also equals to initial heap size of JVM : "+ runtime.totalMemory() / MegaBytes );

System.out.println("Returns the amount of free memory in the JVM : "+ untime.freeMemory() / MegaBytes );

System.out.println(" ===== ----- ===== ");

VolatileTest volatileTest = new VolatileTest();

Thread t1 = new Thread( volatileTest );

t1.start();

Thread t2 = new Thread( volatileTest );

t2.start();

Thread t3 = new Thread( volatileTest );

t3.start();

Thread t4 = new Thread( volatileTest );

t4.start();

Thread.sleep( 10 );;

Thread optimizeation = new Thread() {

@Override public void run() {

System.out.println("Thread Start.");

Integer appendingVal = volatileTest.counter2 + volatileTest.counter2 + volatileTest.counter2;

System.out.println("End of Thread." + appendingVal);

}

};

optimizeation.start();

}

public void sleepThread( long sec ) {

try {

Thread.sleep( sec \* 1000 );

} catch (InterruptedException e) {

e.printStackTrace();

}

}

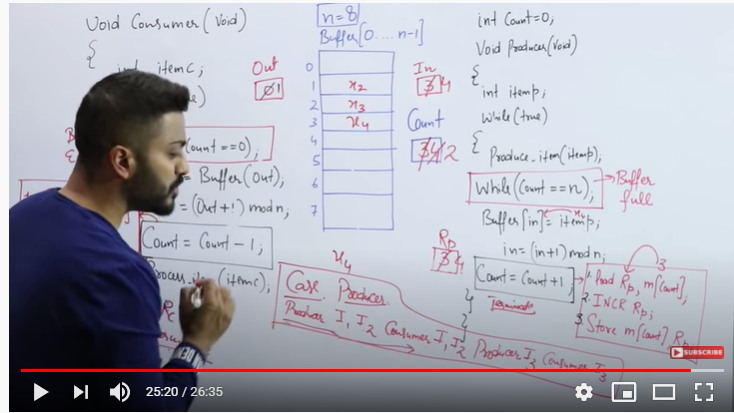
}

Static[Class Field] **vs** Volatile[Instance Field] - Both are not cached by threads

* Static fields are common to all threads and get stored in Method Area. Static with volatile no use. Static field cant be serialized.
* Volatile mainly used with instance variable which get stored in heap area. The main use of volatile is to maintain updated value over all the Threads. instance volatile field can be [Serialized](https://stackoverflow.com/a/31651945/5081877).
  1. Synchronize
  2. Race condition
  3. Deadlocks
  4. BlockingQueue / Producer Consumer problem

I1,I2, Consumer I1, I2, Producer I3,Consumer I3..

Shared memory data mismatch..



Count=count+1;

tempReg, m[count] 🡪 Inc tempReg 🡪 store tempReg to m[count]

* 1. Synchronizers like CyclicBarrier, CountdownLatch

[CountDownLatch](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CountDownLatch.html): Suppose a stone can be lifted by 10 people so you will wait for all 10 to come. Then only you can lift the stone.

[CyclicBarrier](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/CyclicBarrier.html): If you are going to a picnic, and you need to first meet at some common point from where you all will start your journey.

The key difference is that CountDownLatch separates threads into waiters and arrivers while all threads using a CyclicBarrier perform both roles.

* With a latch, the waiters wait for the last arriving thread to arrive, but those arriving threads don't do any waiting themselves.
* With a barrier, all threads arrive and then wait for the last to arrive.

Your latch example implies that all ten people must wait to lift the stone together. This is not the case. A better real world example would be an exam prompter who waits patiently for each student to hand in their test. Students don't wait once they complete their exams and are free to leave. Once the last student hands in the exam (or the time limit expires), the prompter stops waiting and leaves with the tests.

Example :: jmeter.

* 1. Phaser
  2. Atomic classes

1. Java 8 topics
2. Design Patterns / Sorting Algorithms
   1. Singletons

Eager Initialization

Static block initialization

Lazy Initialization

Thread Safe Singleton

Bill Pugh Singleton Implementation

Using Reflection to destroy Singleton Pattern

Enum Singleton

Serialization and Singleton

Singleton pattern restricts the instantiation of a class and ensures that only

one instance of the class exists in the java virtual machine. The singleton

class must provide a global access point to get the instance of the class.

Singleton pattern is used for logging, driver objects, caching and thread

pool.

Singleton design pattern is also used in other design patterns like Abstract

Factory, Builder, Prototype, Facade etc. Singleton design pattern is used in

core java classes also, for example java.lang.Runtime, java.awt.Desktop

* 1. Visitor

**Visitor Pattern** is one of the **behavioral design pattern**. Visitor pattern is used when we have to perform an operation on a group of similar kind of Objects. With the help of visitor pattern, we can move the operational logic from the objects to another class.

For example, think of a Shopping cart where we can add different type of

items (Elements), when we click on checkout button, it calculates the total

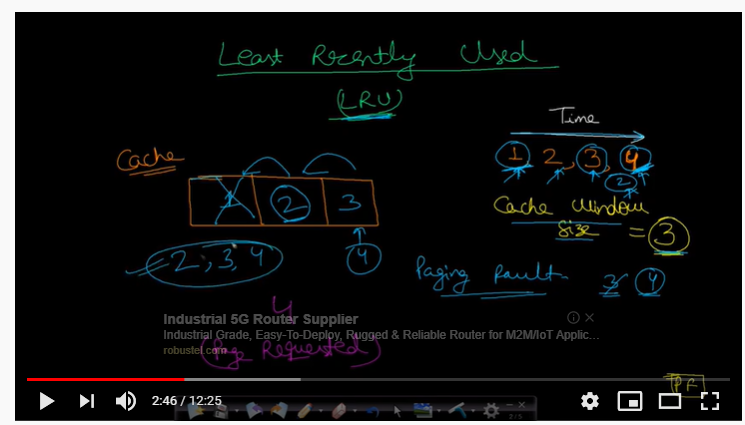
amount to be paid. Now we can have the calculation logic in item classes or

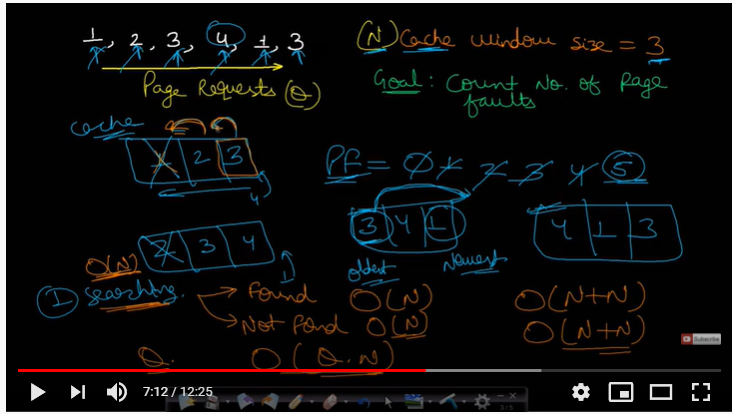
we can move out this logic to another class using visitor pattern

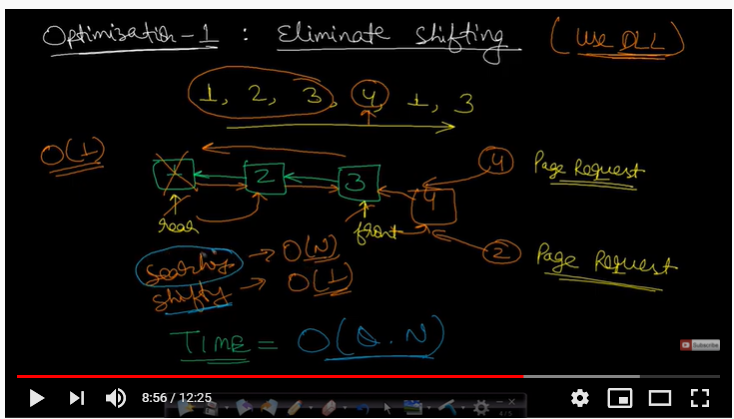
Ex: Shopping cart example with items

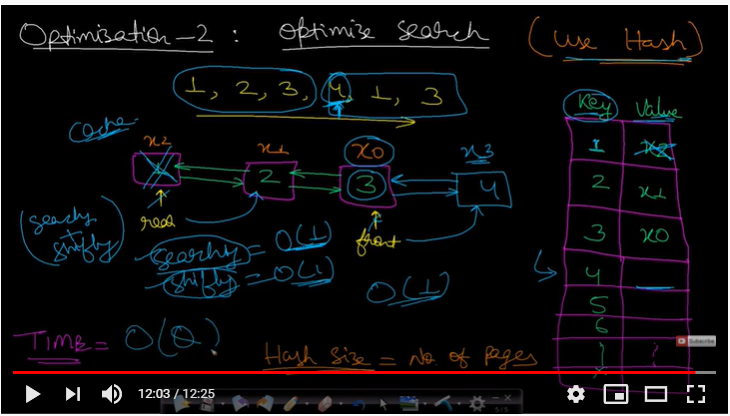
* 1. Template
  2. Decorator
  3. Strategy
  4. Observer
  5. Façade /session Façade
  6. Factory /Abstract Factory
  7. DAO

1. Spring Core
   1. Bean Factory
   2. Application Context
   3. Bean Life Cycle
   4. Init / destroy methods
   5. Bean Listeners
   6. Processors
   7. Scopes
   8. Loading mechanisms
   9. IOC
2. Database (SQL/PLSQL)
   1. DDL
   2. DML
   3. Delete/truncate/Drop
   4. Union / Union All
   5. Index/ clustered- non clustered index (including implementations at DS level)
   6. Procedure
   7. Group by/ having
   8. Count(\*) Max , Avg, etc
   9. Join (types of joins)
   10. Primary Kay / Unique Key
   11. Isolation levels
   12. ACID properties
3. Java Performance Tuning
   1. GC algorithm names only
   2. Heap memory settings
   3. strong, soft, weak and Phantom reference
   4. Stack and Heap Concept
4. Analytical/Logical /Scenario Based questions.
   1. LRU dictionary or Cache









* 1. ATM/Library/HR dept design
  2. Parking allocation
  3. Find most frequently used word from text file
  4. Sorting 10 MB file using 1 MB memory
  5. 1 billion cellphone numbers to finds duplicates
  6. Find duplicate number in Integer Array
  7. Identify palindrome
  8. Fibonacci series printing using recursive
  9. Calculate factorial using recursive and iterative
  10. Implement single elevator , double elevator
  11. Simulate DVD renting system
  12. etc

Sample questions below:

Question Set 1

1. Design a stack that supports getMin() in O(1) time and O(1) extra space.

2. Program for n’th node from the end of a Linked List

3. Semaphore in java 8, print odd and even number using semaphore

4. How ArrayList works internally in Java 8

5. find second largest number in array without sorting in java

6. Sort an array of 0s, 1s and 2s

7. Reverse a linked list

8. Garbage collection algorithms

9. Implement two stacks in an array

10. Producer-Consumer solution using threads in Java

Question Set 2

1. Implement database connection pooling using semaphore

2. Countdown latch/cyclic barrier -explain, difference between cyclic barrier and countdown latch

3. How HashMap works internally in Java 8

4. Function to check if a singly linked list is palindrome

5. Atomic variable -How it works internally

6. Difference between Callable and Runnable

7. Detect and Remove Loop in a Linked List

8. CopyOnWriteArrayList implementation

9. Find first unique character in a String

10. Implement Multithreading application which demonstrates deadlocks and how to avoid deadlocks.

Question Set 3:

1. Find position of an element in a sorted array of infinite numbers

2. How ConcurrentHashMap works internally in Java 8

3. BlockingQueue-Expalin, implement own ArrayBlockingQueue

4. ReentrantLock implementation

5. Intersection point of two Linked Lists.

6. Creating custom exceptions

7. Design a vending machine

8. Java Reference- Soft, Weak, Strong and Phantom

9. Sort an array of 0s, and 1s

10. Different and best approach for Singleton Pattern

Queue Set 4:

1. Search an element in a sorted and rotated array

2. How TreeSet works internally in Java 8

3. UnModifiable collection own implementation

4. Java 8 new features

5. largest-sum-contiguous-subarray

6. Tree traversal with implementation [preorder, postorder, inorder and mirror]

7. Design multi-level parking system

8. Map sort by value

9. Design Principle

10. find the middle element in a linked list

11. Implement StringPool -Flyweight Design Pattern

1 oops/ serialization / Collection (Hash map, hash code, CHM,) imeplemention of hashmap and LRU cache

2  Multithreading ::  was able to answer difference between HM and CHM , what is a deadlock, ExecutorService, Atomic Variable, Blocking Queue, ReEntrantLock, ThreadLocal Class ,CountDownLatch.

3  Sql queries: group by, order by, joins, self- join, inner- join , index, query

4 Spring/ SpringBoot.

5 Data Structure / Problem Solving  : Find Loop inside a Linked list .,Binary Search Tree, find middle element in array

6 Stream Api , Lamda function

7.Explain code for thread Safe SingleTon Class

1.Airport problem, where planes can take off and land . How to manage all when there ll be limitted space at airport technically.

2. Countdownlatch and semaphore

3. Threadsafe classes in java 8

4. Executor service and how it is implemented in your project.

5. Concurrent hashmap, how does it work.

1. Single LinkedList Operations like How to create list, add element to list etc.
2. Exception Handling in your project i.e. how you handled global exception etc.
3. Micro Service Resiliency with the help Hystrix Design pattern.
4. Current project explanation and what tech stack you used in the project.
5. What is Just in Time Compiler and how it helps to improve performance.
6. What is abstraction and where we can use it.
7. Collections like Hashmap and ConncurrentHashMap internal working.
8. Java8 Features such as What is Lybda expression and Funcational Programming.
9. What is the difference between Java5 interface and Java8 Interfaces.
10. You have to filter the employee based on graduation year in a collection using Java8 Streams.

1. Explain the current project and what is your roles and responsibility. What functionalities you developed.
2. Follow up questions from the project it self and questions on Performance improvement.
3. What is Executor Framework and How to create thread pool in Executor Framework
4. Difference between Callable and Runnable.
5. Internal working of Hashmap.
6. What is exception and types of exception. How to create your own exception.
7. Difference between Hashmap and ConcurrentHashmap and when to use which one based on scenarios.
8. How to read a large file in Java and which API you will use to improve performance.

First, if your file contains binary data, then using BufferedReader would be a big mistake (because you would be converting the data to String, which is unnecessary and could easily corrupt the data); you should use a BufferedInputStream instead. If it's text data and you need to split it along linebreaks, then using BufferedReader is OK (assuming the file contains lines of a sensible length).

1. What is Atomic Integer and Use of Volatile variable.
2. What is Future Object and what are its methods available
3. What is Generics and how to use Generics in Future Object.

***Core Java***

1. Define Association and its types i.e. Aggregation and Composition.
   1. With two given classes A & B, how will you implement aggregation and composition programmatically?
2. What do you mean by Immutable classes? Examples from JDK APIs.
   1. Why String is immutable?
   2. Unmodifiable vs Immutable collections.
   3. How can you make immutable classes?
   4. Cross questions on above topic, if non-immutable objects under immutable class and if that non-immutable class further contains non-immutable objects inside it.

🡪 What will be your approach to make the classes immutable in such cases?

1. Can you override instance or static variables? If yes/no, what is the concrete reason behind that?
2. Be prepared with different questions on function overloading. E.g.
   1. Can you overload methods with different return types?
   2. Can you overload with changing the argument types from primitive to wrapper type i.e

Void abc (int a) {}; Void abc (Integer a) {};

* 1. Be prepared with method overloading with Autoboxing and Widening

1. Serliazable vs Externalizable interface with possible cross questions.
2. What are Checked & non-checked exceptions? How you will create runtime exceptions? Explanation with real time scenarios to support the use of both types of exceptions.
3. What is JDK, JRE & JVM with internals of JVM? What are class loaders in java and how can you create your custom class loaders? Troubleshooting approaches for ClassNotFound exception and NoClassDefFoundError.
4. When to use volatile keyword and what is its use?
   1. Does declaring a variable as volatile ensures thread safety? If not then what to use?
5. When to use atomic variables? Which algorithm is used to identify the change? (CAS- Compare & Swap)
6. What is marker interface? Can you declare your own marker interface?
7. How to test a void method?
8. Can an abstract class have a constructor? When is that called?
9. What is difference between init and static block? What is the order of execution among init/static/constructor?

***JDK Memory Model***

1. How is memory managed in JAVA?
2. What is young generation and old generation?
   1. Eden/Survivor Spaces and how the transition happens between the memory spaces.
3. Minor and Major GC?
4. Various Garbage collection types and how to specify which garbage collection should be used?
5. CMS & G1 garbage collection and what does String Deduplication means?
6. How to take heap dump and analyse the same?
7. Concept of String Pool can be touched in order to judge the memory allocation for a String Object.

***Serialization/Externalization***

1. What does serialization means and why is it required?
   1. How can you achieve serialization?
   2. What are compatible and non-compatible changes for a serialized object?
2. What impact does Final, Transient & Static keywords have on member variables during serialization/deserialization?
3. What will happen if one the member of class does not implement Serializable interface?
4. What is the role of serialVersionUID?
5. What will happen if have a collection member variable in a class to be serialized?
6. Does constructor of class gets called during deserialization?
7. How you can avoid Deserialization process creating another instance of Singleton class?
   1. readResolve()

***Collections***

1. How HashSet and TreeSet internally works and what is the complexity to do operations like get, put on these collections?
   1. Will TreeSet allows null objects storage? If yes/no then why?
   2. How TreeSet able to maintain sorting of objects.
2. Scenarios where you want to use ArrayList and LinkedList with reasons.
3. How Collections.sort(list of objects) works internally, what will be the complexity to sort elements using above approach and how?
4. Internal working of HashMap (Any differences in internal implementation w.r.t JDK 1.7 and JDK 1.8), Concurrent HashMap & TreeMap. How LinkedHashMap able to maintain insertion order of elements.
5. Internals of Locking strategy used by Concurrent HashMap to provide synchronized behaviour in concurrent Environment.
6. Understanding of Concurrent APIs like CopyOnWriteArrayList, ConcurrentHashMap etc.
7. How can you sort hashmap on the basis of values with the help of JDK 1.8 Stream APIs and without that?
8. Contract between object’s class equals () & hashcode (). How these methods going to be used inside hashing technique. If you are using any object (like Employee class object) as a custom key inside the HashMap, how you will override these methods?
9. What type of classes should be used as keys for hashmap()?
10. Further questions around this like overriding hashCode() with constant or returning always true/false from equals() method?
11. What is dirty read in a Hashmap?
12. How does rehashing work in a Hashmap?
13. What is fail fast and failsafe?
14. Which collection implementation is failfast and which all are failsafe? (Concurrent modification exception)
15. Which all iterators are available as part of collection API?
16. What add-on feature does list Iterator provides in comparison to other iterator?
17. What are IdentityHashMap and WeakHashMap?

***JDK 1.8 Specific questions***

1. What is a stream? How does it differ from a collection?
2. What is the difference between intermediate, terminal & short-circuit operations?
3. What is the difference between map and flatMap stream operation?
4. What is stream pipelining in Java 8?
5. Program to check even/odd and prime numbers using stream in jdk 1.8
6. What is a functional interface? What are the rules of defining a functional interface?
7. Define default functional interfaces like: Function, Consumer, Supplier , Predicate, BiFunction, BinaryOperator, UnaryOperator
8. What is a lambda expression? What are its advantages? Where do we use a lambda expression?
9. What is a method reference with different types?
10. With interfaces having default methods, how JDK 1.8 able to sort out diamond problem?

***Multithreading & Java Concurrency***

1. What kind of common problems (which usually comes while doing concurrent operations) you have faced in multi-threading environment? How did you resolve it?
2. Mention some guidelines or best practices you have used while writing thread safe code.
3. How do you handle an unhandled exception in the thread?
4. What thread-scheduling algorithm is used in Java?
5. You have thread T1, T2, and T3, how will you ensure that thread T2 run after T1 and thread T3 run after T2?
6. Apart from Thread class instance join (), what are the other ways to do that? How join method is able to achieve it internally?
7. If you have to implement a high-performance cache which allows multiple readers but the single writer to keep the integrity how will you implement it?
8. Thread life cycle with difference between wait, sleep and yield methods.
9. Describe the purpose and use-cases of the fork/join framework.
10. How to generate and analyse Thread Dumps?
11. Difference between object lock and class lock?
12. What will be your design approach, if you have to design your own custom thread pool?
13. What will be your approach to handle uncaught runtime exception generated in run method?
14. What is CountDownLatch & CyclicBarrier? If you have to implement it by your own, what will be your approach?
15. Difference between synchronized and ReentrantLock in java?
16. What is executor framework in java? Explain the usage of Executor, ExecutorService inside that. Explain thread pool configuration in detail like CorePoolSize, MaximumPoolSize and KeepAliveTime.
17. What are the available implementations of ExecutorService in the standard library? What are the benefits of using ThreadPoolExecutor implementation of ExecutorService interface?
18. How static keyword does impacts the thread locks?
19. What is deadlock, livelock & thread starvation?
20. What are Futures object?
21. Completable futures?
22. What is thread local and how to implement it?
23. For which particular use case one should implement a thread local?
24. Blocking queue in JAVA and how that can be implemented?

***Spring***

1. What is Aspect, Advice, Pointcut, JointPoint and Advice Arguments in AOP?
2. What are the different types of Advices?
3. Explain the way you are doing transaction management and error handling in your spring applications.
4. What is the Bean life cycle in Spring Bean Factory Container and what are the callback methods in Spring?
5. What is Spring IoC Container? What is the difference between BeanFactory and ApplicationContext?
6. What is the default scope of beans in Spring? Explain all the scopes available in spring.
7. Difference between singleton scope bean and singleton class?
8. What is dependency injection and the types? When to use Setter and when to use Constructor dependency injection.
9. What do you understand by auto wiring and name the different modes of it?
10. What’s the difference between @Component, @Controller, @Repository & @Service annotations in Spring?
11. Use of @Required, @Autowired, @Resource & @Qualifier annotations
12. How to stop loading some beans in application context at start up?
13. How to resolve circular or cyclic dependency related issues like BeanCurrentlyInCreationException ?
14. Why it’s better to avoid constructor injection if there is any cyclic dependency?
15. How to Inject Prototype Scoped Bean in Singleton Bean so that the injected bean should behave like Prototype instead of outer Singleton bean.
16. Usage of ApplicationContextAware?

***Hibernate***

1. [What are the advantages of Hibernate over JDBC?](https://www.journaldev.com/3633/hibernate-interview-questions-and-answers#hibernate-vs-jdbc)
2. Is [SessionFactory and Session thread safe?](https://www.journaldev.com/3633/hibernate-interview-questions-and-answers#session-factory-thread-safe)
3. How many SessionFactory & Sessions can we have for a connection?
4. [What is hibernate caching? Explain Hibernate first level cache?](https://www.journaldev.com/3633/hibernate-interview-questions-and-answers#hibernate-caching) How to configure second level caching? What is query level cache?
5. Difference between get() & load(), save() & persist(), merge() & update()
6. Different entity states and which operation can be performed in which state?

(Transient/Persistent/Detached)

1. What is HQL, Named SQL query and criteria query? How and when to use which one?
2. [What is cascading and what are different types of cascading?](https://www.journaldev.com/3633/hibernate-interview-questions-and-answers#hibernate-cascading)
3. What is lazy & eager loading in hibernate? What is N+1 SELECT problem in Hibernate?
4. Explain different fetching strategies & Inheritance Mapping Strategies.
5. Which annotation can be used to avoid a field from an entity to be persisted in DB? (@Transient)
6. Have you implemented any connection pooling in your application? If yes, which connection pool you have used? Benefits and drawbacks of using connection pooling?
7. How to resolve LazyInitializationException & OptimisticLockException in Hibernate?
8. What is optimistic and pessimistic locking and which one should be used on scenario basis?
9. What is flush() method and when to use it?
10. How can you define relationships in different entities?
11. OneToOne/OneToMany/ManyToOne/ManyToMany?
12. What is the use of @MappedBy annotation?
13. By default how many tables will get created in different types of entity relationships?
14. How many tables are min required for a ManyToMany relationship?

***Rest:***

1. Difference between @Controller & @RestController?
2. Explain the term ‘Statelessness’ with respect to RESTful WEB service & Enlist advantages and disadvantages of ‘Statelessness’.
3. What are the best practices that are to be followed while designing RESTful web services?
4. What HTTP Status Code 200,201,204,304,400,401,404,409 & 500 states?
5. Explain different HTTP methods like Get, PUT, POST, Delete, Patch, Head & Options?
6. How you are doing API versioning in your application? What are the different approaches available for that with their benefits and drawbacks?
7. How can you make your Rest APIs secure?
8. Which approach you are using to define contract documents of your APIs?
9. What is @RestController?
10. What are the HTTP verbs?
11. What is the difference between POST and PUT?
12. Is it possible to fetch data using POST?
13. How to map a URI to a resource method?
14. What is a content-type?
15. How did you do authentication?
16. How to make a HTTPS URI?
17. Follow-up : What maps the URI to the method?
18. Difference between SOAP and REST
19. When to choose REST
20. What is the response code used indicate the successful processing of request.
21. What approach should be used while designing URI’s? Design one for GET/PUT/POST/DELETE.
22. How can you identify DDOS attack and what approach will you apply in order to handle the DDOS attack for your rest service?
23. Basics of any of security concepts like OAuth 2 & JWT?

***Database:***

1. How to optimize the performance of DB queries? What is query execution plan? How can you generate it?
2. What are indices? Advantages?
3. Is it good to have too many indices?
4. What is primary key?
5. Some SQL query to fetch the names of department and the count from the table where the employees’ count is more than 10.
6. How primary key, unique key and foreign keys are different.
7. How does index work.
8. What is the best data type to use as an index.
9. Difference between Stored Proc and Functions
10. What is Normalization? Why use normalization?
11. What are SQL joins and their types?
12. Delete, Truncate vs Drop command

***Design Pattern:***

1. Have you used any Creational, Structural or Behavioural design pattern in your application? Explain the use case as per your application?
2. With use case, explain at least one design pattern from Creational, Structural & Behavioural types.
3. Best way to implement Singleton Design Pattern.
4. What are different design principles? Can you specify any 4 design principles apart from the SOLID design principles?
5. Give example of decorator design pattern in Java? Does it operate on object level or class level?
6. What is façade design pattern and its usage?
7. What is flyweight design pattern and where is it used in JAVA API? (String Memory allocation).

Apart from the above topics, be prepared on JMS*(e.g. Durable vs Non-Durable Topics etc)*, Caching *(InMemory cache like Redis, MemCache & Distributed Cache)*, Profiling & Performance tuning at code, DB as well as JVM level*(i.e. defining proper heap size, efficient GC etc)*.

Knowledge about the Build tool like maven and continuous integration like Jenkins/TeamCity should be brushed up.

1. Proper sequence :: ({[]}) and ([}{]
2. Trie data structure application program:: to generate the gift coupon for mobile numbers starting with sequence
3. Generate the random value from collection, it should not follow any pattern //

Random r = new Random();

r.nextInt(upperRange);

1. Swagger
2. Params and Headers in Web calls
3. Sort map by value
4. Patterns
5. Java 8 methods ::
6. Stream in java 8
7. Feign client
8. Spring Cloud consul
9. Quartz
10. Thread pool
11. Callable interface
12. Marker interface
13. If 2 classes are inter dependent, how bean creation will happen, setter injection and constructor injection
14. dependency injection vs inversion of control
15. two data source in spring ?
16. @LoadBalanced ?
17. @Value
18. @SpringBootApplication ?