

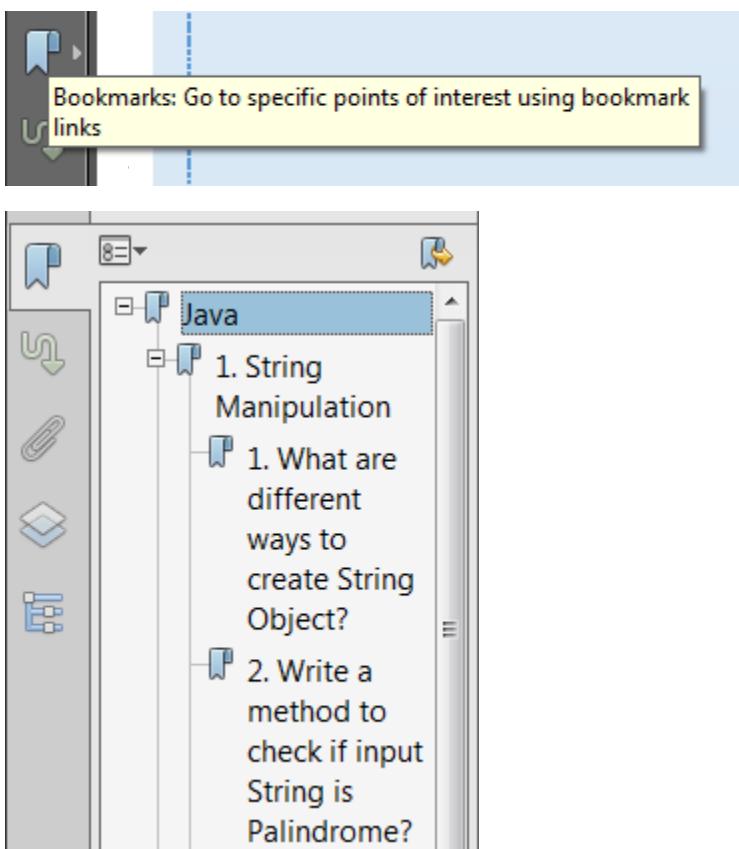
Quick and most likely asked interview
questions guide for IT professionals
with skill set: Java, OOP and Data
structures

Interview Guide

Questions likely fired in
interview...

Akshay Chincholkar
www.linkedin.com/in/akshaychincholkar
Updated: 23th July, 2018

Note: Use Bookmark Pane for convenient navigation in PDF:



For word doc please enable navigation pane:

To open the Navigation pane, press Ctrl+F, or click View > Navigation Pane.

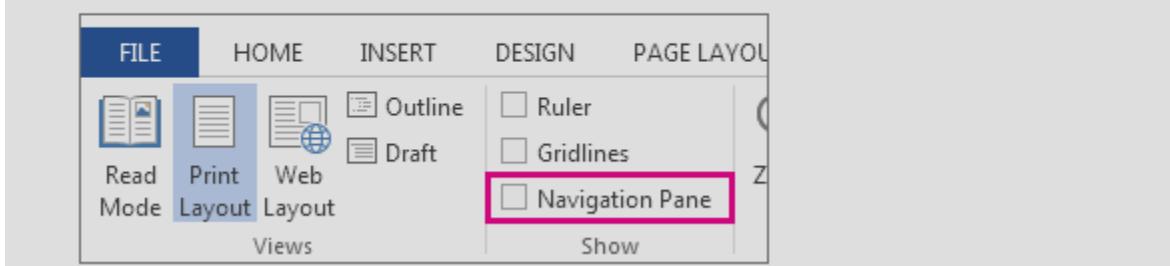


Table of Contents

Java	20
1. String Manipulation	20
1. What are different ways to create String Object?	20
2. Write a method to check if input String is Palindrome?	20
3. Write a method that will remove given character from the String?	21
4. What is String subSequence method?	22
5. Java String subSequence Example.....	22
6. How to compare two Strings in java program?.....	23
7. How to convert String to char and vice versa?.....	24
8. How to convert String to byte array and vice versa?	25
9. Can we use String in switch case?	26
10. Difference between String, StringBuffer and StringBuilder?	28
11. WAP to find number Alphabets, Numbers and Special characters are present inside the given string.	28
12. WAP to create an immutable class.....	30
2. Exception Handling	35
1. What do you mean by the checked and the unchecked exceptions in java?	35
2. Explain exception hierarchy in java?	38
3. What are 5 exception handling keywords in java?.....	38
4. Explain what is Error in java?	39
5. What are differences between Exception and Error in java?	39
6. How to create user defined checked and unchecked Exception in java?	41
7. How to create user defined checked and unchecked Exception in java?	42
8. Is it allowed to use multiple catch block in java?	42
9. What is Automatic resource management in java 7?	44
10. Explain try-with-resource in java?	46
11. Now, question comes why we need not to close file when we are using Try-with-resources in java?.....	47
12. Discuss which checked and unchecked exception can be thrown/declared by subclass method while overriding superclass method in java?.....	47
13. What will happen when catch and finally block both return value, also when try and finally both return value in java?.....	48

14.	What is exception propagation in java?.....	49
15.	Can a catch or finally block throw exception in java?	52
16.	Why shouldn't you use Exception for catching all exceptions in java?	52
17.	What is Difference between multiple catch block and multi catch syntax?	52
18.	can a method be overloaded on basis of exceptions in java ?	55
19.	What are the differences between between ClassNotFoundException and NoClassDefFoundError in java ?	56
2.1	NoClassDefFoundError is a Error in java. Error and its subclasses are regarded as unchecked exceptions in java.	56
2.2	2	57
2.3	Here is the hierarchy of java.lang.ClassNotFoundException -	57
2.4	Here is the hierarchy of java.lang.NoClassDefFoundError -	57
20.	What are the most important frequently occurring Exception and Errors which you faced in java?	57
21.	What is stackTrace in exception handling?	58
3.	Serialization	59
1.	What is Serialization in java?.....	59
2.	How do we Serialize object, write a program to serialize and deSerialize object and persist it in file (Important)?	59
3.	How can you customize Serialization and DeSerialization process when you have implemented Serializable interface (Important)?	60
4.	Wap to explain how can we Serialize and DeSerialize object by implementing Externalizable interface (Important)?	61
5.	How can you avoid certain member variables of class from getting Serialized?	62
6.	What is serialVersionUID?	62
7.	What will be impact of not defining serialVersionUID in class (Important)?	63
8.	What are compatible and incompatible changes in Serialization process?	63
9.	What if Serialization is not available, is any other alternative way to transfer object over network?	64
10.	Why static member variables are not part of java serialization process (Important)?	64
11.	What is significance of transient variables?	64
12.	What will happen if one the member of class does not implement Serializable interface (Important)?	65
13.	What will happen if we have used List, Set and Map as member of class?	65

14.	Is constructor of class called during DeSerialization process?.....	65
15.	Are primitive types part of serialization process?	65
16.	What values will int and Integer will be initialized to during DeSerialization process if they were not part of Serialization?	65
17.	What is singleton?	65
18.	What happens when we serialize the singleton?	67
4.	Collections	69
1.	Can we use custom object as key in HashMap? If yes then how?	69
2.	Why do we need to override equals and hashCode method?.....	69
3.	Why to override hashCode method?.....	70
4.	Why to override equals method?.....	71
5.	If two objects have same hashCode, are they always equal?.....	71
6.	If two objects equals() method return true, do objects always have same hashCode? ...	71
7.	What classes should i prefer to use a key in HashMap?	72
8.	Can overriding of hashCode() method cause any performance issues?.....	73
9.	What are subinterfaces of Collection interface in java? Is Map interface also a subinterface of Collection interface in java?	73
10.	What are differences between ArrayList and LinkedList in java?	74
11.	What are differences between List and Set interface in java?	76
12.	What are differences between Iterator and ListIterator? in java	78
13.	What are differences between Collection and Collections in java?	80
14.	What are core classes and interfaces in java.util.List hierarchy in java?	81
15.	What are core classes and interfaces in java.util.Set hierarchy?	81
16.	What are differences between Iterator and Enumeration in java?	82
17.	What are differences between HashMap and Hashtable in java?	83
18.	when to use HashSet vs LinkedHashSet vs TreeSet in java?	85
19.	What are differences between HashMap and ConcurrentHashMap in java?	86
20.	When to use HashMap vs Hashtable vs LinkedHashMap vs TreeMap in java?	90
21.	What are differences between HashMap vs IdentityHashMap in java?	92
22.	What is WeakHashMap in java?	94
23.	What is the difference between fail fast and fail safe iterators?	94
5.	Multithreading.....	97
1.	What is Thread in java?.....	97

2.	What is difference between Process and Thread in java?	97
3.	How to implement Threads in java?	98
4.	We should implement Runnable interface or extend Thread class. What are differences between implementing Runnable and extending Thread?.....	98
5.	How can you say Thread behaviour is unpredictable? (Important)	100
6.	When threads are not lightweight process in java?	100
7.	How can you ensure all threads that started from main must end in order in which they started and also main should end in last? (Important)	100
8.	Write a program to demonstrate the join()	102
9.	What are the versions of join() method?	103
10.	What is difference between starting thread with run() and start() method? (Important) 105	
11.	What is significance of using Volatile keyword? (Important).....	107
9.	Can we have volatile methods in java?	111
10.	Can we have synchronized variable in java?.....	111
11.	Can you again start Thread?	111
12.	What is race condition in multithreading and how can we solve it? (Important)....	111
13.	What is deadlock in multithreading? Write a program to form DeadLock in multi threading and also how to solve DeadLock situation. What measures you should take to avoid deadlock? (Important).....	112
14.	Why wait(), notify() and notifyAll() are in Object class and not in Thread class? (Important).....	114
15.	Is it important to acquire object lock before calling wait(), notify() and notifyAll()? 115	
16.	Have you ever generated thread dumps or analyzed Thread Dumps? (Important) ..	115
17.	What is life cycle of Thread, explain thread states? (Important).....	116
18.	Are you aware of preemptive scheduling and time slicing?.....	117
22.	What are daemon threads?	117
23.	Why suspend() and resume() methods are deprecated?	119
24.	Why destroy() methods is deprecated?.....	120
25.	As stop() method is deprecated, How can we terminate or stop infinitely running thread in java? (Important)	120
26.	what is significance of yield() method, what state does it put thread in?	121
27.	What is significance of sleep() method in detail, what state does it put thread in ?	122

28.	Difference between wait() and sleep() ? (Important)	123
29.	Does thread leaves object lock when wait() method is called?	124
30.	What will happen if we don't override run method?.....	124
5.1	When we call start() method on thread, it internally calls run() method with newly created thread. So, if we don't override run() method newly created thread won't be called and nothing will happen.	124
31.	What will happen if we override start method?.....	125
32.	Can we acquire lock on class? What are ways in which you can acquire lock on class? 125	
33.	Difference between object lock and class lock?	126
34.	Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time?	128
35.	Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time? 129	
36.	Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter static synchronized method2() at same time?	
	131	
37.	Suppose you have thread and it is in synchronized method and now can thread enter other synchronized method from that method?	134
38.	Suppose you have thread and it is in static synchronized method and now can thread enter other static synchronized method from that method?.....	135
39.	Suppose you have thread and it is in static synchronized method and now can thread enter other non static synchronized method from that method?	136
40.	Suppose you have thread and it is in synchronized method and now can thread enter other static synchronized method from that method?.....	138
41.	Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time? 140	
42.	Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time?	141
43.	Difference between wait() and wait(long timeout), What are thread states when these method are called?	144
44.	Can a constructor be synchronized?	144
45.	Can you find whether thread holds lock on object or not?	145
46.	. What do you mean by thread starvation?	145

47.	What is addShutdownHook method in java?	146
48.	How you can handle uncaught runtime exception generated in run method?.....	146
49.	What is ThreadGroup in java, What is default priority of newly created threadGroup, mention some important ThreadGroup methods ?	149
50.	What are thread priorities?	149
51.	Output question 1.....	150
52.	Output question 2.....	152
53.	Output question 3.....	153
54.	Output question 4.....	155
55.	Output question 5.....	156
56.	Output question 6.....	159
57.	Output question 7.....	160
58.	Output question 8.....	162
59.	Output question 9.....	164
60.	Output question 10.....	165
61.	Output question 11.....	167
62.	Output question 12.....	169
63.	Output question 13.....	170
64.	Output question 14.....	171
65.	Output question 15.....	174
66.	Output question 16.....	175
67.	Output question 17.....	176
68.	Output question 18.....	177
69.	Output question 19.....	178
70.	Output question 20.....	180
71.	Output question 21.....	182
72.	Question 82. Output question 22.....	183
73.	Deadlock in Java Multithreading.....	184
6.	Concurrency	193
1.	What is ThreadPool?	193
2.	What is ThreadFactory? Why is it implemented?	193
3.	What is ThreadGroup? Why is it used?	194
4.	Explain about the shutDownHook in java.....	195

5.	Difference between call() and the run() methods?	196
6.	What is Java.util.concurrent.CyclicBarrier ?	197
7.	Important point of CyclicBarrier in Java.....	203
8.	What is executor framework in java?.....	204
9.	What are differences between execute() and submit() method of executor framework in java?.....	204
10.	What is Semaphore in java 7?	205
11.	How can you implement Producer Consumer pattern using Semaphore in java?....	207
12.	How can you implement your own Semaphore?.....	212
13.	What is significance of atomic classes in java 7?.....	222
14.	What are Future and Callable? How are they related in java?	223
15.	Similarity and differences between java.util.concurrent.Callable and java.lang.Runnable in java?.....	224
16.	What is CountDownLatch in java?.....	224
17.	Where can you use CountDownLatch in real world?.....	226
18.	How can you implement your own CountDownLatch in java?	226
19.	What is CyclicBarrier in java?.....	234
20.	Why is CyclicBarrier cyclic in java?	235
21.	Where could we use CyclicBarrier in real world?.....	235
22.	How can you implement your own CyclicBarrier in java?	235
6.1	Answer. It is very complex thread concurrency interview question.Even most of the experienced developers are not aware of this question. Please read ReentrantLock class provides implementation of Lock's newCondition() method in java - description and solving producer consumer program using this method.	246
6.2	Read more about Fork/Join Framework - Parallel programming in java.....	248
6.3	Answer. Developers must have knowledge of atomic operations in thread concurrency java. Java provides some classes in java.util.concurrent.atomic which offers an alternative to the other synchronization in java.	257
6.4	Please see Atomic operations in java.....	257
7.	Garbage Collection	265
1.	Give digramatic JVM Heap memory (Hotspot heap structure).	265
2.	What is Throughput in gc(garbage collection) in java ?	265
3.	What are pauses in gc(garbage collection) in java?	265
4.	JVM Heap memory (Hotspot heap structure) in java consists of which elements?...	266

5.	What is Young Generation (Minor garbage collection occurs in Young Generation)?	266
6.	What is Old Generation or (tenured generation) - (Major garbage collection occurs in Old Generation)?	267
7.	What is Permanent Generation or (Permgen) - (full garbage collection occurs in permanent generation in java)?.....	267
8.	What are the most important VM (JVM) PARAMETERS in JVM Heap memory? ...	268
9.	What are parameters for Young Generation(VM PARAMETERS for Young Generation) ?	269
10.	What are the parameters for the old Generation (tenured) - (VM PARAMETERS for Old Generation) ?	270
11.	What are the parameters for Permanent Generation (VM PARAMETERS for Permanent Generation)?	270
12.	Name the other important VM (JVM) parameters for java heap in java.....	271
13.	What are the different Garbage collectors in java?	271
14.	What is Serial collector / Serial GC (Garbage collector) ?.....	272
15.	What is Throughput GC (Garbage collector) or Parallel collector in java?	272
16.	What is Incremental low pause garbage collector (train low pause garbage collector) in java?.....	274
17.	What is Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java? 275	
18.	What is G1 Garbage Collector (or Garbage First) in java?	278
19.	Difference between Serial GC (Garbage collector) vs Throughput GC (Garbage collector) in java?	281
20.	What is ParNew collector ?	281
21.	What is Automatic Garbage Collection in JVM heap memory in java?	282
22.	How garbage collection is done using Marking and deletion in java?	282
23.	Very important points about GC (Garbage Collection) in Java	284
24.	Summary of garbage collection in java-.....	287
8.	Miscellaneous Topics.....	293
8.1	What is significance of final in java?	293
8.2	What is difference between using instanceof operator and getClass() in equals method?	293
8.3	What is Immutable class?.....	293
8.4	Java Garbage Collection.....	294
8.5	Ways to create an object of a class?	296

8.6	What are the types of references in Java?.....	301
8.7	What are ‘Strong References’?	301
8.8	What are ‘Weak References’?	302
8.9	What are ‘Soft References’?	304
8.10	What are ‘Phantom/Ghost References’?.....	306
8.11	Difference between FileReader and BufferedReader in java file IO.....	308
8.12	Why do we assign a parent reference to the child object in Java?	310
OOP.....		331
1.	SOLID principles	332
2.	Association.....	332
3.	Inheritance.....	335
4.	Aggregation.....	343
5.	Composition.....	347
6.	What is Abstraction.....	353
7.	What is Encapsulation?.....	355
8.	Difference between Abstraction and Encapsulation ?	358
9.	Difference between Data hiding and Abstraction?	359
10.	Polymorphism	360
10.1	Rules for method overriding:	360
Design Patterns		369
1.	Creational Design Patterns.....	369
1.1	Singleton Pattern.....	370
1.2	Factory Pattern.....	376
1.3	Abstract Factory Pattern	378
1.4	Builder Pattern	382
1.5	Prototype Pattern	384
2.	Structural Design Patterns.....	386
2.1	Adapter Design Pattern.....	386
2.2	Composite Pattern.....	390
2.3	Proxy Pattern	393
2.4	Flyweight Pattern.....	394
2.5	Facade Pattern.....	398

2.6	Bridge Pattern	401
2.7	Decorator Pattern	403
3.	Behavioral Design Patterns	406
3.1	Template Method Pattern	406
3.2	Mediator Pattern	409
	Data Structures	413
1.	Arrays.....	413
1.1	Find a pair in an array of size 'n', whose sum is X	413
1.2	Find a majority element in an array of size 'n'	414
1.3	Find the number occuring odd number of times in a given array of size 'n'.....	414
1.4	Algorithm to reverse an array	414
1.5	Algorithm to rotate array of size 'n' by 'd' elements	415
1.6	Algorithm to segregate 0's and 1's in an array	415
1.7	Find the maximum difference between two elements such that larger element appears after the smaller element.....	416
1.8	Algorithm to merge an array of size 'n' into another array of size 'm+n'.....	416
1.9	Algorithm to find two repeating numbers in a given array	417
1.10	Algorithm to find duplicate elements in O(n) time and O(1) extra space, for a given array of size 'n'	418
1.11	Find the index in an array such that the sum of elements at lower indices is equal to the sum of elements at higher indices.....	418
1.12	Algorithm to find the maximum difference of j - i such that a[j] > a[i], for a given an array of 'n' elements	418
1.13	Algorithm to find the triplet whose sum is X	418
1.14	Algorithm to find a sub array whose sum is X	418
1.15	Algorithm to find the largest sub array with equal number of 0's and 1's	418
1.16	Algorithm to find the number of triangles that can be formed with three different array elements as three sides of triangles, for a given unsorted array of n elements	418
1.17	Algorithm to find the smallest integer value that can't be represented as sum of any subset of a given array.....	418
1.18	Algorithm to find the common element in given three sorted arrays	418
1.19	Algorithm to find the contiguous sub-array with maximum sum, for a given array of postive and negative numbers.....	418

1.20 Given an array of integers, sort the array into a wave like array and return it. (arrange the element into a sequence such that $a_1 \geq a_2 \leq a_3 \geq a_4 \leq a_5 \dots$ etc.....	418
1.21 Algorithm to find the next greater number formed after permuting the digits of given number	418
1.22 Algorithm to find the sum of bit difference in all pairs that can be formed from array of n elements.....	418
1.23 Trapping rain water problem	418
1.24 Algorithm to find the minimum number of platforms required for the railway station so that no train waits according to arrival and departure time.....	418
1.25 Rotate 2-Dimentional array	419
1.26 Lock and Key problem	419
1.27 Rearrange an array so that $a[i]$ becomes $a[a[i]]$ with $O(1)$ extra space	419
1.28 Traverse a matrix of integers in spiral form	419
1.29 Given an array consisting 0's, 1's and 2's, write a algorithm to sort it.....	419
1.30 Given a positive number X, print all jumping numbers(all adjacent digits in it differ by 1) smaller than or equal to X	419
1.31 Given an array and an integer 'k', find the maximum, for each and every contiguous subarray of size 'k'	419
1.32 Search an element in a sorted rotated array.....	419
1.33 Find the maximum value of $a[j]-a[i]+a[l]-a[k]$, for every four indices i, j, k, l such that $i < j < k < l$	419
2. Linked List.....	Error! Bookmark not defined.
2.1 Algorithm to find the nth node from end of the linked list.....	419
2.2 Algorithm to find the middle node in a linked list	419
2.3 Algorithm to find the intersection point of two linked lists	419
2.4 Reversal of linked list	419
2.5 Algorithm to detect loop in linked list.....	419
2.6 Algorithm to find starting node of a loop in a linked list	419
2.7 Algorithm to check given linked list is palindrome (or) not	419
2.8 Algorithm to reverse alternative K nodes in a single linked list	419
2.9 Algorithm to clone a linked list with next and random pointer are given...many more	
419	
3. Stack.....	Error! Bookmark not defined.
3.1 Reversal of a stack	420

3.2	Algorithm to find next greater element on the right side of an array.	420
3.3	Implementation of the following operations in stack in O(1) time. Push(), pop(), isEmpty(), isFull() and getMin().	420
3.4	Algorithm to find the celebrity in minimum number of questions in a party.....	420
3.5	Algorithm to the stock span problem is a financial problem where we have a series of 'n' daily price for a stock and we need to calculate span of stock's price for all n days.....	420
3.6	Algorithm to merge overlapping intervals.....	420
3.7	Find the largest rectangular area possible in a given histogram.....	420
3.8	Given an integer array of size 'n', find the maximum of the minimum's of every window size in the array.....	420
3.9	Calculate minimum number of bracket reversals needed to make an expression balanced.	420
3.10	Design a stack, to find getmin() in O(1) time and O(1) space complexity.....	420
3.11	Find if an expression has duplicate or not....many more	420
4.	Queues.....	Error! Bookmark not defined.
4.1	Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.	420
4.2	Implement LRU Cache.....	420
4.3	Find the first circular tour that visits all petrol pumps.....	420
4.4	Find the largest multiple of 3.....	420
5.	Trees.....	Error! Bookmark not defined.
5.1	Implement in order traversal without stack and recursion	420
5.2	Convert a binary tree into its mirror tree	420
5.3	Check if a given binary tree is sum tree or not	421
5.4	Determine if the given two trees are identical or not	421
5.5	Print out all of its root to leaf paths in a given binary tree	421
5.6	Find a lowest common ancestor of a given two nodes in a abinary search tree.....	421
5.7	Find a lowest common ancestor of a given two nodes in a binary tree	421
5.8	Level order traversal in spiral form	421
5.9	Convert an arbitrary binary tree to a tree that holds children sum property	421
5.10	Find the Diameter of a BST.....	421
5.11	Construct tree from given inorder and post order traversal.....	421
5.12	Convert a Binary Tree to a circular DLL	421
5.13	Evaluation of expression tree	421

5.14	Print extreme node of each level of Binary Tree in alternative order	421
5.15	Print cousins of a given node in Binary Tree	421
5.16	Diagonal traversal of Binary Tree	421
5.17	Construct tree from ancestor matrix	421
5.18	Given a Binary Tree, find vertical sum of the nodes that are in same vertical line..	421
5.19	Find multiplication of sums of data of leaves at same level.....	421
5.20	Given a binary tree, find maximum value we can get by subtracting value of node B from value of node A.....	421
5.21	Print nodes in a top view of Binary Tree.....	421
5.22	Given a Binary Tree and a number k, remove all nodes that lie only on root to leaf path(s) of length smaller than k.....	421
5.23	Serialize and deserialize an N-ary tree.	421
5.24	Reverse alternate levels of a perfect Binary Tree.....	421
5.25	Print all nodes that are at distance k from a leaf node.....	421
5.26	Custom tree problem.	422
5.27	Construct complete binary tree from its linked list representation.....	422
5.28	Find next right nodes of given leafs in a binary tree.	422
5.29	Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root.....	422
5.30	Convert a given tree to its sum tree.	422
5.31	Given a binary tree, find out if the tree can be folded or not.....	422
5.32	Find largest sub tree having identical left and right sub tree.....	422
5.33	Convert a normal binary search tree to balanced BST.	422
5.34	Check if removing an edge can divide a binary tree in the form of n-ary tree.....	422
5.35	locking and unlocking of resource arranged on the form of n-ary tree.	422
6.	Heaps.....	Error! Bookmark not defined.
6.1	Find K largest (or smallest) elements in array.....	422
6.2	Tournament tree method using binary heap	422
6.3	Find a Median in a stream of integers.	422
6.4	Sort a nearly sorted array(or k sorted).	422
6.5	Given array representation of min Heap, convert it to max Heap.....	422
6.6	Check if a given binary tree is Heap.....	422
6.7	Find kth largest element in a stream.	422

6.8	Print all elements in sorted order from row and column wise sorted matrix.....	422
6.9	Given n ropes of different length, connect with minimum cost.	422
6.10	Given k sorted arrays of size n each, merge them.	422
6.11	Design an efficient data structure for given operations find min(), findmax(), deletemin(), Insert(),delete().....	422
7.	Strings	Error! Bookmark not defined.
7.1	Find a maximum occurring character in the input string.....	Error! Bookmark not defined.
7.2	Remove all duplicates from a given string.	Error! Bookmark not defined.
7.3	A program to check if strings are rotations of each other or not.	Error! Bookmark not defined.
7.4	Find the smallest window in a string containing all characters of another string	Error! Bookmark not defined.
7.5	Revere words in a given string.	Error! Bookmark not defined.
7.6	Find all distinct palindromic sub strings of a given string	Error! Bookmark not defined.
7.7	Remove all adjacent duplicate characters in a string... Error! Bookmark not defined.	
7.8	Given a string, find the Run length encoding of given string.... Error! Bookmark not defined.	
7.9	Check whether two strings are anagram of each other or not. .. Error! Bookmark not defined.	
7.10	Find the first non-repeating character from a stream of characters .. Error! Bookmark not defined.	
7.11	Given an array of strings , find if the string can be of characters..... Error! Bookmark not defined.	
7.12	Find a excel column name from a given column number. Error! Bookmark not defined.	
7.13	Convert one string to another using minimum number of given operation..... Error! Bookmark not defined.	
7.14	Check if a given sequence of moves for a robot is circular (or) not. Error! Bookmark not defined.	
7.15	Print concatenation of zig-zag string in 'n' rows..... Error! Bookmark not defined.	
7.16	Minimum number of palindromic sub sequence to be removed to empty a binary string. Error! Bookmark not defined.	
7.17	All combinations of string that can be used to dial a number. .. Error! Bookmark not defined.	

8. Graphs	Error! Bookmark not defined.
8.1 Given a directed graph $G=(V,E)$, find whether G has a cycle.....	423
8.2 Given an undirected graph $G=(V,E)$, find whether G has a cycle or not.	423
8.3 Given a directed graph $G=(V,E)$, find whether there is path between two vertices v_i and v_j .423	
8.4 Given a 2D boolean matrix, find the number of islands.....	423
8.5 Given a connected undirected graph, find all the articulation points	423
8.6 Given an undirected graph, find all the bridges in the graph.	423
8.7 Given a directed graph, find all the strongly connected components.....	423
8.8 Given a weighted directed acyclic graph, find the shortest path from a vertex to all the other vertices.....	423
8.9 Given a weighted directed acyclic graph, find the longest path from a vertex to all the other vertices.....	423
8.10 Check whether a given graph is bipartite or not.	423
8.11 Find number of connections a person till nth level.	423
9. Bit Manipulation	Error! Bookmark not defined.
Algorithm.....	425
1. Sorting Algorithms.....	426
1.1 Bubble Sort	426
1.2 Selection Sort.....	427
1.3 Insertion Sort	427
1.4 Merge Sort	428
1.5 Quick sort	430
1.6 Heap Sort	431
2. Divide & Conquer.....	432
2.1 Find the median of two sorted arrays	432
2.2 Count inversions in an array	432
2.3 Find majority Element in a sorted array	432
2.4 Find the maximum and minimum of an array using minimum number of comparisons	
432	
2.5 The skyline problem	432
2.6 Given two binary strings that represent value of two integers, find the product of two strings.....	432
2.7 Given an array of integers. Find a peak element in it.....	432

2.8	Find the missing number in Arithmetic Progression	432
2.9	Given an array of n points in the plane, find out the closest pair of points in the array. 432	
3.	Back Tracking.....	432
3.1	Print all permutations of a given string.	432
3.2	Find subset of elements that are selected from a given set whose sum adds up to a given number K.	432
3.3	Given a set of n integers, divide the set in two subsets of $n/2$ sizes each such that the difference of the sum of two subsets is as minimum as possible.	432
3.4	Solve Sudoku using backtracking.....	432
3.5	Given a maze, NxN matrix. A rat has to find a path from source to destination. Left top corner is the source and right bottom corner is destination. There are few cells which are blocked, means rat can-not enter into those cells.	432
4.	Pattern searching	433
4.1	Given a text and a pattern, find all occurrences of pattern in a given text. Using naive approach.....	433
4.2	Given a text and a pattern, find all occurrences of pattern in a given text. Using Rabin-Karp algorithm.....	433
4.3	Given a text and a pattern, find all occurrences of pattern in a given text. Using Finite automata approach.....	433
4.4	Given a text and a pattern, find all occurrences of pattern in a given text. Using Boyer moore algorithm.	433
4.5	Given a text and a pattern, find all occurrences of pattern in a given text. Using KMP algorithm.....	433
4.6	Given a string, find the longest sub string which is palindrome using manacher's algorithm.....	433
4.7	Find all occurrences of a given word in a matrix.	433
5.	Greedy Algorithms.....	433
5.1	Given an array of jobs with different time intervals. Find the minimum time to finish all jobs.....	433
5.2	Given a universe of n elements, collection of subsets. Find a minimum cost sub collection that covers all elements.....	433
5.3	Given n cities and distances between every pair of cities, select k cities to place warehouses, such that the maximum distance of a city to a warehouse is minimized.	433
6.	Dynamic Programming.....	434

- 6.1 Find the length of the longest sub sequence of a given sequence such that all elements of the sub sequence are sorted in increasing order 434
- 6.2 Given two sequences, find the length of longest sub sequence present in both of them. 434
- 6.3 Given a cost matrix and a position (m, n) , Find cost of minimum cost path to reach (m, n) from (0, 0) 434
- 6.4 Coin change problem..... 434
- 6.5 Find the length of the longest palindrome sub sequence..... 434
- 6.6 Find th sum of maximum sum sub sequence of the given array 434
- 6.7 You have a rectangular grid of dimension 2 x n. You need to find out the maximum sum such that no two chosen numbers are adjacent , vertically, diagonally (or) horizontally. 434
- 6.8 Given an array A with n elements and array B with m elements. With m you have to insert (n-m) zero's in between array B such that the dot product of array A and array B is maximum 434
- 6.9 Transform a string into palindrome on removing at most k characters from it..... 434
- 6.10 Find the longest even length sub string such that sum of first and second half is same.. 434
- 6.11 Count number of ways to reach a given score in a game. 434
- 6.12 Compute sum of digits in all number from 1 to n. 434
- 6.13 Collect maximum points in a grid using two traversals 434
- 6.14 Given a 2xn board and titles of size 2x1, count the number of ways to tile he given board using the 2x1 tiles..... 434
- 6.15 Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true..... 434
- 6.16 Given a Binary Tree, find size of the Largest Independent Set(LIS) in it..... 434
- 6.17 There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time. Count the number of ways, the person can reach the top. 435
- 6.18 Find total number of non-decreasing numbers with n digits. 435
- 6.19 Egg dropping problem. 435
- 6.20 Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces..... 435

6.21 Given N jobs where every job is represented by Start Time, Finish Time, Profit or Value Associated. Find the maximum profit subset of jobs such that no two jobs in the subset overlap.....	435
6.22 Box stacking problem.....	435
6.23 Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.....	435
6.24 Given a binary matrix, find out the maximum size square sub-matrix with all 1s... ..	435
6.25 Find the maximum coins you can collect by bursting the balloons wisely.	435
Databases	435
1. - SQL Queries Basics.....	445
2. - SELECT statement and syntax	445
3. - SQL Data Types	445
4. - WHERE operators: comparison operators, LIKE, BETWEEN, IN	445
5. - Scalar Functions	445
6. - Aggregate Functions (SUM, COUNT, MIN, MAX, AVG)	445
7. - GROUP BY	445
8. - ORDER BY	445
9. - JOINS (INNER JOIN, RIGHT and LEFT JOIN, OUTER JOIN, CROSS JOIN)	445
10. - Set Theory (INTERSECT, UNION, MINUS)	445
11. - Subqueries.....	445
12. - INSERT, UPDATE, DELETE.....	445
13. - DML vs DDL vs DCL	445
14. - DDL: CREATE, DROP, ALTER	445
15. - Constraints	445
16. - Indexes	445
17. - DCL privileges: GRANT, REVOKE.....	445
18. - Transactions	445
19. - Views	445
20. - Triggers	445
21. - Cursors	445

Java

String Manipulation

1. What are different ways to create String Object?

We can create String object using `new` operator like any normal java class or we can use double quotes to create a String object. There are several constructors available in String class to get String from char array, byte array, StringBuffer and StringBuilder.

```
String str = new String("abc");
String str1 = "abc";
```

When we create a String using double quotes, JVM looks in the String pool to find if any other String is stored with same value. If found, it just returns the reference to that String object else it creates a new String object with given value and stores it in the String pool.

When we use `new` operator, JVM creates the String object but don't store it into the String Pool. We can use `intern()` method to store the String object into String pool or return the reference if there is already a String with equal value present in the pool.

2. Write a method to check if input String is Palindrome?

A String is said to be Palindrome if it's value is same when reversed. For example “aba” is a Palindrome String.

String class doesn't provide any method to reverse the String but `StringBuffer` and `StringBuilder` class has reverse method that we can use to check if String is palindrome or not.

```
private static boolean isPalindrome(String str) {
    if (str == null)
        return false;
    StringBuilder strBuilder = new StringBuilder(str);
    strBuilder.reverse();
    return strBuilder.toString().equals(str);
}
```

Sometimes interviewer asks not to use any other class to check this, in that case we can compare characters in the String from both ends to find out if it's palindrome or not.

```
private static boolean isPalindromeString(String str) {
    if (str == null)
        return false;
    int length = str.length();
    System.out.println(length / 2);
    for (int i = 0; i < length / 2; i++) {
        if (str.charAt(i) != str.charAt(length - i - 1))
            return false;
    }
    return true;
}
```

3. Write a method that will remove given character from the String?

We can use `replaceAll` method to replace all the occurrence of a String with another String. The important point to note is that it accepts String as argument, so we will use `Character` class to create String and use it to replace all the characters with empty String.

```
private static String removeChar(String str, char c) {
    if (str == null)
        return null;
    return str.replaceAll(Character.toString(c), "");
}
```

4. What is String subSequence method?

Java 1.4 introduced CharSequence interface and String implements this interface, this is the only reason for the implementation of subSequence method in String class. Internally it invokes the String substring method.

5. Java String subSequence Example

Java 1.4 introduced CharSequence interface and String implements this interface, this is the only reason for the implementation of subSequence method in String class. Internally it invokes the [String substring](#) method.

```
public CharSequence subSequence(int beginIndex, int endIndex) {
    return this.substring(beginIndex, endIndex);
}
```

String subSequence method returns a character sequence that is a subsequence of this sequence. An invocation of this method of the form `str.subSequence(begin, end)` behaves in exactly the same way as the invocation of `str.substring(begin, end)`.

Below is a simple java String subSequence method example.

`StringSubsequence.java`

```
package com.journaldev.examples;

public class StringSubsequence {

    /**
     * This class shows usage of String subSequence method
     *
     * @param args
     */
    public static void main(String[] args) {
```

```

        String str = "www.journaldev.com";

        System.out.println("Last 4 char String: " +
str.subSequence(str.length() - 4, str.length()));

        System.out.println("First 4 char String: " +
str.subSequence(0, 4));

        System.out.println("website name: " +
str.subSequence(4, 14));

        // substring vs subSequence

        System.out.println("substring == subSequence ? " +
(str.substring(4, 14) == str.subSequence(4, 14)));

        System.out.println("substring equals subSequence ? " +
+ (str.substring(4, 14).equals(str.subSequence(4, 14))));

    }

}

}

```

Output of the above String subSequence example program is:

```

Last 4 char String: .com

First 4 char String: www.

website name: journaldev

substring == subSequence ? false

substring equals subSequence ? true

```

There is no benefit in using `subSequence` method, ideally you should always use `String substring` method.

6. How to compare two Strings in java program?

Java String implements `Comparable` interface and it has two variants of `compareTo()` methods.

`compareTo(String anotherString)` method compares the String object with the String argument passed lexicographically. If String object precedes the argument passed, it returns negative integer and if String object follows the argument String passed, it returns positive integer. It returns zero when both the String have same value, in this case `equals(String str)` method will also return true.

`compareTolgnoreCase(String str)`: This method is similar to the first one, except that it ignores the case. It uses String CASE_INSENSITIVE_ORDER Comparator for case insensitive comparison. If the value is zero then `equalsIgnoreCase(String str)` will also return true.

7. How to convert String to char and vice versa?

This is a tricky question because String is a sequence of characters, so we can't convert it to a single character. We can use use `charAt` method to get the character at given index or we can use `toCharArray()` method to convert String to character array.

String class has three methods related to char. Let's look at them before we look at a java program to convert string to char array.

1. `char[] toCharArray()`: This method converts string to character array. The char array size is same as the length of the string.
2. `char charAt(int index)`: This method returns character at specific index of string. This method throws `StringIndexOutOfBoundsException` if the index argument value is negative or greater than the length of the string.
3. `getChars(int srcBegin, int srcEnd, char dst[], int dstBegin)`: This is a very useful method when you want to convert part of string to character array. First two parameters define the start and end index of the string; the last character to be copied is at index `srcEnd-1`. The characters are copied into the char array starting at index `dstBegin` and ending at `dstBegin + (srcEnd-srcBegin) - 1`.

Let's look at a simple string to char array java program example.

```
package com.journaldev.string;

public class StringToCharJava {
    public static void main(String[] args) {
        String str = "journaldev";
    }
}
```

```

//string to char array

char[] chars = str.toCharArray();

System.out.println(chars.length);

//char at specific index

char c = str.charAt(2);

System.out.println(c);

//Copy string characters to char array

char[] chars1 = new char[7];

str.getChars(0, 7, chars1, 0);

System.out.println(chars1);

}

}

```

In above program, `toCharArray` and `charAt` usage is very simple and clear.

In `getChars` example, first 7 characters of str will be copied to chars1 starting from it's index 0.

That's all for converting string to char array and string to char java program.

8. How to convert String to byte array and vice versa?

We can use String `getBytes()` method to convert String to byte array and we can use String constructor `new String(byte[] arr)` to convert byte array to String.

```

package com.journaldev.util;

import java.util.Arrays;

public class StringByteArray {

    /**
     * This class shows how to convert String to byte array and
     * byte array to String in java
     * @param args
     */
    public static void main(String[] args) {
        String str = "www.journaldev.com";

```

```

    //convert String to byte array
    byte[] byteArr = str.getBytes();
    System.out.println("String to byte array : "+Arrays.toString(byteArr));
    //convert byte array to String
    String str1 = new String(byteArr);
    System.out.println("byte array to String : "+str1);
    //let's see if str and str1 are equals or not
    System.out.println("str == str1? " + (str == str1));
    System.out.println("str.equals(str1)? " + (str.equals(str1)));
}

}

```

String class method `getBytes()` returns the **byte array from String** and String constructor can be used to create **String from byte array** in java.

Output of the above program is:

```

1  String to byte array : [119, 119, 119, 46, 106, 111, 117, 114, 110, 97, 108, 100, 101,
2  byte array to String : www.journaldev.com
3  str == str1? false
4  str.equals(str1)? true

```

9. Can we use String in switch case?

This is a tricky question used to check your knowledge of current Java developments. Java 7 extended the capability of switch case to use Strings also, earlier java versions doesn't support this.

If you are implementing conditional flow for Strings, you can use if-else conditions and you can use switch case if you are using Java 7 or higher versions.

Keys points to know for java switch case String are:

1. Java switch case String make code more readable by removing the multiple if-else-if chained conditions.
2. Java switch case String is case sensitive, the output of example confirms it.
3. Java Switch case uses `String.equals()` method to compare the passed value with case values, so make sure to add a NULL check to avoid `NullPointerException`.
4. According to [Java 7 documentation for Strings in Switch](#), java compiler generates more efficient byte code for String in Switch statement than chained if-else-if statements.
5. Make sure to use java switch case String only when you know that it will be used with Java 7 else it will throw Exception.

Thats all for Java switch case String example.

Tip: We can use **java ternary operator** rather than switch to write smaller code.

```
package com.journaldev.util;

public class SwitchStringExample {

    public static void main(String[] args) {
        printColorUsingSwitch("red");
        printColorUsingIf("red");
        // switch case string is case sensitive
        printColorUsingSwitch("RED");
        printColorUsingSwitch(null);
    }

    private static void printColorUsingIf(String color) {
        if (color.equals("blue")) {
            System.out.println("BLUE");
        } else if (color.equals("red")) {
            System.out.println("RED");
        } else {
            System.out.println("INVALID COLOR CODE");
        }
    }

    private static void printColorUsingSwitch(String color) {
        switch (color) {
            case "blue":
                System.out.println("BLUE");
            case "red":
```

```

        break;

    case "red":

        System.out.println("RED");

        break;

    default:

        System.out.println("INVALID COLOR CODE");

    }

}

```

10. Difference between String, StringBuffer and StringBuilder?

String is immutable and final in java, so whenever we do String manipulation, it creates a new String. String manipulations are resource consuming, so java provides two utility classes for String manipulations - StringBuffer and StringBuilder.

StringBuffer and StringBuilder are mutable classes. StringBuffer operations are thread-safe and synchronized where StringBuilder operations are not thread-safe. So when multiple threads are working on same String, we should use StringBuffer but in single threaded environment we should use StringBuilder.

StringBuilder performance is fast than StringBuffer because of no overhead of synchronization.

11. WAP to find number Alphabets, Numbers and Special characters are present inside the given string.

```

String s1 = "I l@ve M@rgan & Stanley!";
int characters = 0;
int numbers = 0;
int specialChars = 0;
System.out.println(specialChars);
for(int i=0;i<s1.length();i++) {
    char c = s1.charAt(i);
    System.out.println("Character at "+i+" is "+c);
    if((c>='a' && c<='z') || (c>='A' && c<='Z')) {
        System.out.println("It is the character");
        characters++;
    }
}

```

```
        }else if(c>='0' && c<='9') {  
    System.out.println("Is the digit");  
    numbers++;  
}else{  
    System.out.println("It is the special  
character");  
    specialChars++;  
}  
}  
System.out.println("Characters: "+characters+ "  
digits:"+numbers+ " Special Chars:"+specialChars);
```

Output: Character at 0 is I
It is the character
Character at 1 is
It is the special character
Character at 2 is 1
Is the digit
Character at 3 is @
It is the special character
Character at 4 is v
It is the character
Character at 5 is e
It is the character
Character at 6 is
It is the special character
Character at 7 is M
It is the character
Character at 8 is @
It is the special character
Character at 9 is r
It is the character
Character at 10 is g
It is the character
Character at 11 is a
It is the character
Character at 12 is n
It is the character
Character at 13 is
It is the special character

```
Character at 14 is &
It is the special character
Character at 15 is
It is the special character
Character at 16 is $
It is the special character
Character at 17 is t
It is the character
Character at 18 is a
It is the character
Character at 19 is n
It is the character
Character at 20 is l
It is the character
Character at 21 is e
It is the character
Character at 22 is y
It is the character
Character at 23 is !
It is the special character
Characters: 14 digits:1 Special Chars:9
```

12. WAP to create an immutable class.

```
public final class FinalClassExample {

    private final int id;

    private final String name;

    private final HashMap<String, String> testMap;

    public int getId() {
        return id;
    }
}
```

```
public String getName() {  
    return name;  
}  
  
/**/  
 * Accessor function for mutable objects  
 */  
public HashMap<String, String> getTestMap() {  
    //return testMap;  
    return (HashMap<String, String>) testMap.clone();  
}  
  
/**/  
 * Constructor performing Deep Copy  
 * @param i  
 * @param n  
 * @param hm  
 */  
  
public FinalClassExample(int i, String n,  
HashMap<String, String> hm) {  
    System.out.println("Performing Deep Copy for Object  
initialization");  
    this.id=i;  
    this.name=n;
```

```

        HashMap<String, String> tempMap=new
HashMap<String, String>();

        String key;

        Iterator<String> it = hm.keySet().iterator();

        while(it.hasNext()) {

            key=it.next();

            tempMap.put(key, hm.get(key));

        }

        this.testMap=tempMap;

    }

}

/**
 * Constructor performing Shallow Copy
 * @param i
 * @param n
 * @param hm
 */
public FinalClassExample(int i, String n,
HashMap<String, String> hm) {

    System.out.println("Performing Shallow Copy for
Object initialization");

    this.id=i;

    this.name=n;

    this.testMap=hm;

}

```

```
*/  
  
/**  
 * To test the consequences of Shallow Copy and how to  
 avoid it with Deep Copy for creating immutable classes  
 * @param args  
 */  
  
public static void main(String[] args) {  
  
    HashMap<String, String> h1 = new  
    HashMap<String, String>();  
  
    h1.put("1", "first");  
  
    h1.put("2", "second");  
  
    String s = "original";  
  
    int i=10;  
  
    FinalClassExample ce = new  
    FinalClassExample(i,s,h1);  
  
    //Lets see whether its copy by field or reference  
    System.out.println(s==ce.getName());  
    System.out.println(h1 == ce.getTestMap());  
    //print the ce values  
    System.out.println("ce id:"+ce.getId());  
    System.out.println("ce name:"+ce.getName());  
    System.out.println("ce testMap:"+ce.getTestMap());
```

```

//change the local variable values

i=20;

s="modified";

h1.put("3", "third");

//print the values again

System.out.println("ce id after local variable
change:"+ce.getId());

System.out.println("ce name after local variable
change:"+ce.getName());

System.out.println("ce testMap after local variable
change:"+ce.getTestMap());


HashMap<String, String> hmTest = ce.getTestMap();

hmTest.put("4", "new");

System.out.println("ce testMap after changing
variable from accessor methods:"+ce.getTestMap());


}

}

```

Output:

```

Performing Deep Copy for Object initialization

true

false

ce id:10

ce name:original

```

```
ce testMap:{2=second, 1=first}  
ce id after local variable change:10  
ce name after local variable change:original  
ce testMap after local variable change:{2=second, 1=first}  
ce testMap after changing variable from accessor  
methods:{2=second, 1=first}
```

Exception Handling

1. What do you mean by the checked and the unchecked exceptions in java?

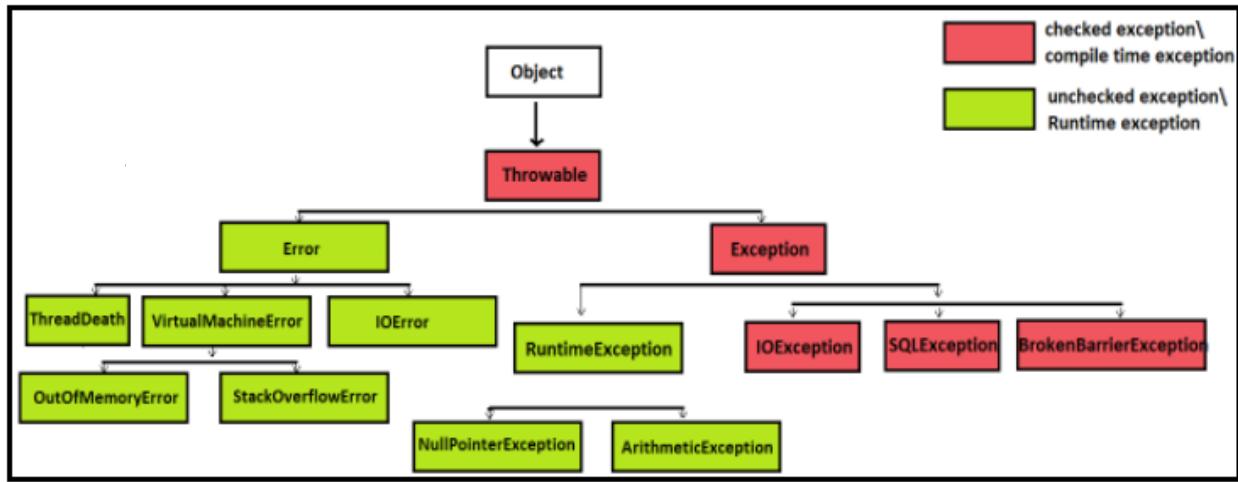
In **Java exceptions** under Error and RuntimeException classes are **unchecked exceptions**, everything else under throwable is **checked**. Consider the following **Java** program. It compiles fine, but it throws ArithmeticException when run. The compiler allows it to compile, because ArithmeticException is an **unchecked exception**.

	Property	checked exception	unchecked exception
1	Also known as	checked exceptions are also known as compileTime exceptions in java.	unchecked exceptions are also known as runtime exceptions in java.
2	Should be solved at compile or runtime?	Checked exceptions are those which need to be taken care at compile time in java.	Unchecked exceptions are those which need to be taken care at runtime in java.
3	Benefit/ Advantage	We cannot proceed until we fix compilation issues which are most likely to happen in program, this helps us in avoiding runtime problems upto lot of extent in java.	Whenever runtime exception occurs execution of program is interrupted, but by handling these kind of exception we avoid such interruptions and end up giving some meaningful message to user in java.
4	Creating custom/own exception	<pre>class UserException extends Exception { UserException(String s) { super(s); } }</pre> <p>By extending <code>java.lang.Exception</code>, we can create checked exception.</p>	<pre>class UserException extends RuntimeException { UserException(String s) { super(s); } }</pre> <p>By extending <code>java.lang.RuntimeException</code>, we can create unchecked exception.</p>

Activate Wind

5	Exception propagation	For propagating checked exceptions method must throw exception by using throws keyword.	unchecked exceptions are automatically propagated in java.
6	handling checked and unchecked exception while overriding superclass method	<p><i>If superclass method throws/declare checked exception ></i></p> <ul style="list-style-type: none"> • overridden method of subclass can declare/throw narrower (subclass of) checked exception (As shown in Program), or • overridden method of subclass cannot declare/throw broader (superclass of) checked exception (As shown in Program), or • overridden method of subclass can declare/throw any unchecked /RuntimeException (As shown in Program) 	<p><i>If superclass method throws/declare unchecked ></i></p> <ul style="list-style-type: none"> • overridden method of subclass can declare/throw any unchecked /RuntimeException (superclass or subclass) (As shown in Program), or • overridden method of subclass cannot declare/throw any checked exception (As shown in Program),
	Which classes are which type of exception? either checked or unchecked exception?	The class Exception and all its subclasses that are not also subclasses of RuntimeException are checked exceptions in java.	The class RuntimeException and all its subclasses are unchecked exceptions. Likewise, The class Error and all its subclasses are unchecked exceptions in java.
7	Most frequently faced exceptions	SQLException , IOException , ClassNotFoundException	NullPointerException , ArithmaticException ArrayIndexOutOfBoundsException .

2. Explain exception hierarchy in java?



`java.lang.Object` is superclass of all classes in java.

`java.lang.Throwable` is superclass of `java.lang.Exception` and `java.lang.Error`

`java.lang.Exception` is superclass of `java.lang.RuntimeException`, `IOException`, `SQLException`, `BrokenBarrierException` and many more other classes in java.

`java.lang.RuntimeException` is superclass of `java.lang.NullPointerException`, `ArithmaticException` and many more other classes in java.

`java.lang.Error` is superclass of `java.lang.VirtualMachineError`, `IOError`, `AssertionError`, `ThreadDeath` and many more other classes in java.

`java.lang.VirtualMachineError` is superclass of `java.lang.OutOfMemoryError`, `StackOverflowError` and many more other classes in java.

3. What are 5 exception handling keywords in java?

Answer. This is another very important exception handling interview question in java.

5 keyword in java exception handling in java

- **try** - Any exception occurring in try block is catched by catch block.
- **catch** - catch block is always followed by try block in java.
- **finally** **finally block** can only exist if try or try-catch block is there, finally block can't be used alone in java.

[**Features of finally >**](#)

- finally block is **always executed** irrespective of exception is thrown or not.
- finally is **keyword** in java.
- finally block is optional in java, we may use it or not.

*finally block is **not executed** in following scenarios >*

- finally is not executed when **System.exit** is called.
- if in case **JVM crashes** because of some java.util.**Error**.

- **throw** throw is a **keyword** in java.
- **throw** keyword allows us to throw **checked or unchecked exception**.
- **throws** throws is written in method's definition to indicate that method can throw **exception** in java.

4. Explain what is Error in java?

Answer. Experienced developers must know in detail about Exception handling interview question in java. [**java.lang.Error**](#)

- Error is a subclass of **Throwable** in java.
- Error **indicates some serious problems** that our **application should not try to catch in java**.
- Errors are **abnormal conditions in application**.
- Error and its subclasses are regarded as **unchecked exceptions** in java

Must know :

[ThreadDeath](#) is an error which application must not try to catch but it is normal condition in java.

5. What are differences between Exception and Error in java?

Answer. It is another very important exception interview question to differentiate between Exception and Error in java.

	Property	Exception	Error
--	----------	----------------------------------	------------------------------

1	serious problem?	Exception does not indicate any serious problem.	Error indicates some serious problems that our application should not try to catch.
2	divided into <u>checked</u> and <u>unchecked</u>	Exception are divided into checked and unchecked exceptions in java.	Error are not divided further into such classifications in java.
3	Which classes are which type of exception? either checked or unchecked exception?	The class Exception and all its subclasses that are not also subclasses of RuntimeException are checked exceptions. The class RuntimeException and all its subclasses are unchecked exceptions. Likewise, The class Error and all its subclasses are unchecked exceptions in java.	Error and its subclasses are regarded as unchecked exceptions in java
4	Most frequently faced exception and errors	checked exceptions> SQLException, IOException, ClassNotFoundException unchecked exceptions> <u>NullPointerException</u> , ArithmaticException,	VirtualMachineError, IOException, AssertionError, ThreadDeath, OutOfMemoryError, StackOverflowError.
5	Why to catch or not to catch?	Application must catch the Exception because they does not cause any major threat to application in java.	Application must not catch the Error because they does cause any major threat to application.

		<p>Example ></p> <p>Let's say errors like OutOfMemoryError and StackOverflowError occur and are caught then JVM might not be able to free up memory for rest of application to execute, so it will be better if application don't catch these errors and is allowed to terminate in java.</p>
--	--	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

6. How to create user defined checked and unchecked Exception in java?

Answer. Very important exception handling interview question.

Interviewers generally expects interviewees to write code to create checked and unchecked Exception in java.

Creating user defined checked exception in java >

```
class UserDefinedException extends Exception {
    UserDefinedException(String s) {
        super(s);
    }
}
```

By extending `java.lang.Exception`, we can create checked exception.

Creating user defined unchecked exception in java >

```
class UserDefinedException extends RuntimeException {
    UserDefinedException(String s) {
        super(s);
    }
}
```

```
}
```

By extending `java.lang.RuntimeException`, we can create unchecked exception.

7. How to create user defined checked and unchecked Exception in java?

Answer. Very important exception handling interview question.

Interviewers generally expects interviewees to write code to create checked and unchecked Exception in java.

[Creating user defined checked exception in java >](#)

```
class UserDefinedException extends Exception {
    UserDefinedException(String s) {
        super(s);
    }
}
```

By extending `java.lang.Exception`, we can create checked exception.

[Creating user defined unchecked exception in java >](#)

```
class UserDefinedException extends RuntimeException {
    UserDefinedException(String s) {
        super(s);
    }
}
```

By extending `java.lang.RuntimeException`, we can create unchecked exception.

8. Is it allowed to use multiple catch block in java?

Answer. Another exception handling interview question which will test your practical knowledge and understanding of Exception handling in java. [Java exception handling](#) allows us to use [multiple catch block](#) in java.

Important Point about multiple catch block in java >

1. **Exception class handled in starting catch block must be subclass of Exception class handled in following catch blocks (otherwise we will face compilation error).**
2. Either one of the multiple catch block will handle exception at time in java.

Program - Let's understand the concept of multiple catch block in java>

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

public class ExceptionTest {

    public static void main(String[] args) {

        try{
            int i=10/0; //will throw ArithmeticException
        }catch(ArithmetiException ae){
            System.out.println("Exception handled - ArithmetiException");
        }catch(RuntimeException re){
            System.out.println("Exception handled - RuntimeException");
        }catch(Exception e){
            System.out.println("Exception handled - Exception");
        }
    }
}

/*OUTPUT
Exception handled - ArithmetiException
*/
```

In the above above >

ArithmetiException has been used in **first** catch block

RuntimeException has been used in **second** catch block

Exception has been used in **third** catch block

Exception is superclass of **RuntimeException** and

RuntimeException is superclass of **ArithmaticException**.

9. What is Automatic resource management in java 7?

Answer. Java provides a feature to make the code more robust and to cut down the lines of code. This feature is known as Automatic Resource Management(ARM) using **try-with-resources** from Java 7 onwards. The try-with-resources statement is a try statement that declares one or more resources.

This statement ensures that each resource is closed at the end of the statement which eases working with external resources that need to be disposed or closed in case of errors or successful completion of a code block.

What is a resource?

A resource is an object that must be closed after the program is finished using it. Any object that implements `java.lang.AutoCloseable`, which includes all objects which implement `java.io.Closeable`, can be used as a resource.

In try-with-resources method there is no use of finally block. the file resource is opened in try block inside small brackets. Only the objects of those classes can be opened within the block which implements AutoCloseable interface and those object should also be local. The resource will be closed automatically regardless of whether try statement completes normally or abruptly.

Syntax:

The following example reads the first line from a file. It uses an instance of `BufferedReader` to read data from the file. `BufferedReader` is a resource that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException
{
    try (BufferedReader br = new BufferedReader(new FileReader(path)))
    {
        return br.readLine();
    }
}
```

Example:

```
// Java program to illustrate
// Automatic Resource Management
// in Java without finally block

class Resource {

    public static void main(String args[])
    {
        String str = "";
        try (BufferedReader br = new BufferedReader(new FileReader("file.txt")))
        {
            str = br.readLine();
        }
    }
}
```

```

BufferedReader br = null;
System.out.println("Enter the file path");
br = new BufferedReader(new InputStreamReader(System.in));
try {
    str=br.readLine();
} catch(IOException e) {
    e.printStackTrace();
}
// try with Resource
// note the syntax difference
try (BufferedReader b = new BufferedReader(new FileReader(str))) {
    String s;
    while ((s = b.readLine()) != null) {
        System.out.println(s);
    }
}
catch (IOException e) {
    e.printStackTrace();
}
}
}

```

Features of multi catch syntax in java >

- Has **improved way of catching multiple exceptions.**
- This syntax does **not looks clumsy in java.**
- **Reduces developer efforts** of writing multiple catch blocks in java.
- Allows us to **catch more than one exception in one catch block.**

Here is the **multi catch syntax** >

```
try{
    //code . . . .
}catch(IOException | SQLException ex){
    //code . . . .
}
```

We could separate different exceptions using **pipe (|)** in java.

10. Explain try-with-resource in java?

Answer. Again experienced java developers must be well versed with this exception interview question. **Before java 7**, we used to write **explicit code for closing file in finally block by using try-finally block** like this >

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

public class TryWithResourceTest {
    public static void main(String[] args) throws IOException {
        InputStream inputStream = null;
        try{
            inputStream = new FileInputStream("c:/txtFile.txt");
            //code.....
        }finally{
            if(inputStream!=null)
                inputStream.close();
        }
    }
}
```

In java 7, using Try-with-resources >

- we need not to write **explicit code for closing file**.

```
import java.io.FileInputStream;
```

```

import java.io.IOException;
import java.io.InputStream;
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class TryWithResourseTest {
    public static void main(String[] args) throws IOException {
        try (InputStream inputStream = new
FileInputStream("c:/txtFile.txt")) {
            //code...
        }
    }
}

```

Using multiple resources inside Try-with-resources is also allowed in java.

11. Now, question comes why we need not to close file when we are using Try-with-resources in java?

Answer. Again experienced java developers must be well versed with this exception interview question. Because **FileInputStream** implements **java.lang.AutoCloseable interface** (**AutoCloseable** interface's close method automatically closes resources which are no longer needed) in java.

Which classes can be used inside Try-with-resources in java?

All the classes which implements **AutoCloseable** interface can be used inside **Try-with-resources in java**.

12. Discuss which checked and unchecked exception can be thrown/declared by subclass method while overriding superclass method in java?

Answer. It's very very important exception handling interview question. Experienced and freshers all must be able to answer this question.

*If superclass method throws/declare **unchecked/RuntimeException** in java >*

- overridden method of subclass **can declare/throw any unchecked /RuntimeException (superclass or subclass)**, or
- overridden method of subclass **cannot declare/throw any checked exception in java**, or
- overridden method of subclass **can declare/throw same exception in java**, or
- overridden method of subclass **may not declare/throw any exception in java**.

*If superclass method throws/declare **checked/compileTime exception in java** >*

- overridden method of subclass **can declare/throw narrower** (subclass of) **checked exception**, or
- overridden method of subclass **cannot declare/throw broader** (superclass of) **checked exception**, or
- overridden method of subclass **can declare/throw any unchecked /RuntimeException**, or
- overridden method of subclass **can declare/throw same exception**, or
- overridden method of subclass **may not declare/throw any exception in java**.

*If superclass method does **not throw/declare any exception in java** >*

- overridden method of subclass **can declare/throw any unchecked /RuntimeException** , or
- overridden method of subclass **cannot declare/throw any checked exception**, or
- overridden method of subclass **may not declare/throw any exception in java**.

13. What will happen when catch and finally block both return value, also when try and finally both return value in java?

Answer. This is very important exception handling interview question for experienced developers.

When **catch and finally block** both return value, **method will ultimately return value returned by finally block** irrespective of value returned by **catch** block.

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

public class ExceptionTest {

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

```

        System.out.println("method return -> "+m());
    }

    static String m(){
        try{
            int i=10/0; //will throw ArithmeticException
        }catch(ArithmaticException e){
            return "catch";
        }finally{
            return "finally";
        }
    }
}

/*OUTPUT
method return -> finally
*/

```

In above program, `i=10/0` will throw `ArithmaticException` and enter catch block to return `"catch"`, but ultimately control will enter finally block to return `"finally"`.

Likewise, when **try and finally block** both return value, **method will ultimately return value returned by finally block** irrespective of value returned by try block

14. What is exception propagation in java?

Answer.

Experienced developers must know in detail about Exception handling interview question in java. Even freshers must try and understand this in depth concept of exception propagation in java.

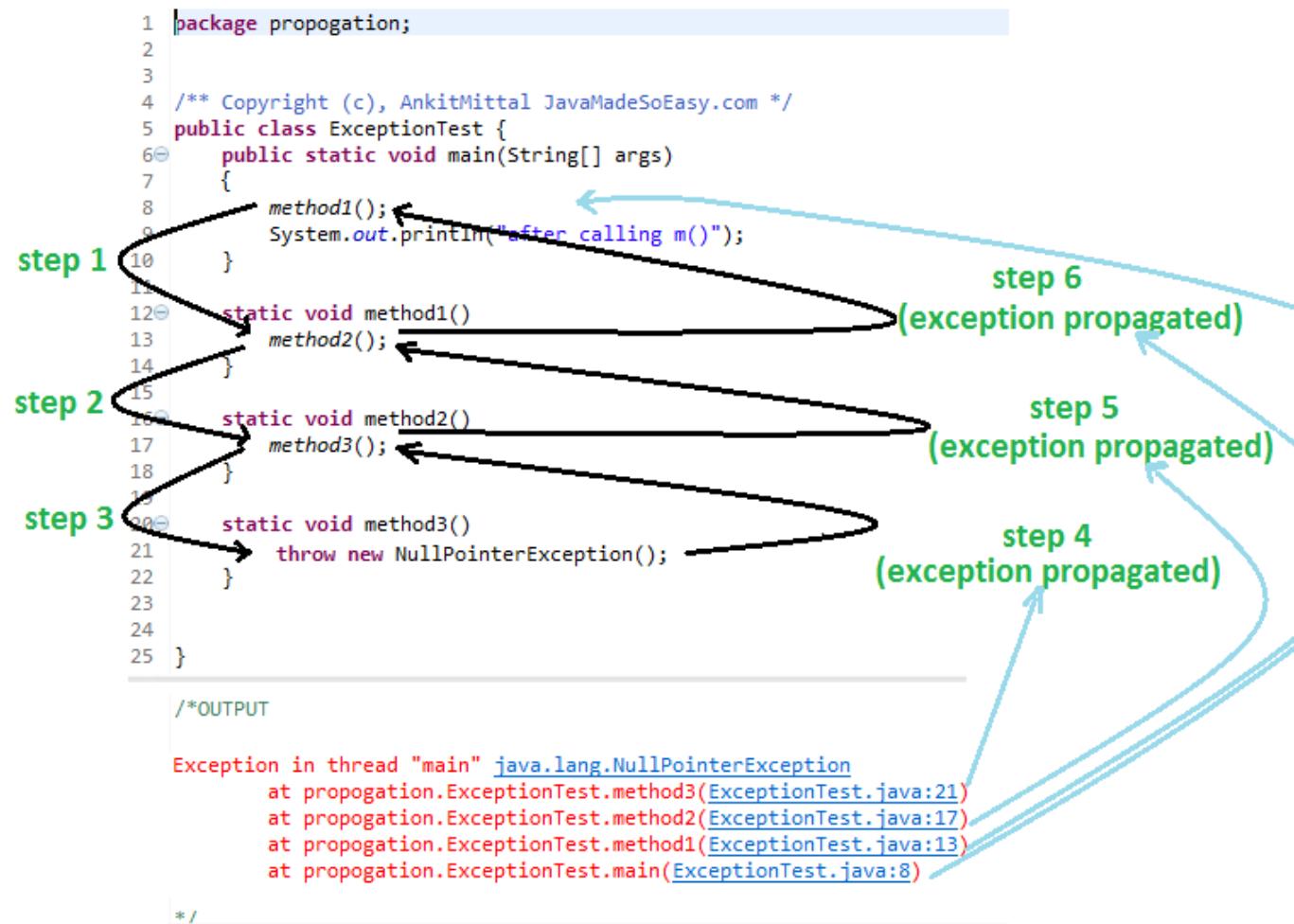
Whenever methods are called **stack** is formed and an exception is first thrown from the top of the stack and if it is not caught, it starts coming down the stack to previous methods until it is not caught.

If exception remains uncaught even after reaching bottom of the stack it is propagated to JVM and program is terminated in java.

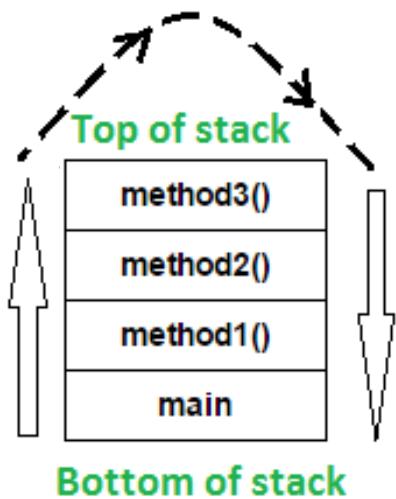
Propagating unchecked exception (NullPointerException)

>

unchecked exceptions are automatically propagated in java.



stack of methods is formed >



In the above program, stack is formed and an exception is first thrown from the top of the stack [**method3()**] and it remains uncaught there, and starts coming down the stack to previous methods to **method2()**, then to **method1()**, than to **main()** and it remains uncaught throughout.

exception remains uncaught even after reaching bottom of the stack [**main()**] so it is propagated to JVM and ultimately program is terminated by throwing exception [as shown in output] in java.

*Propagating **checked** exception (*FileNotFoundException*) using **throws** keyword >*

For propagating **checked** exceptions method must throw exception by using **throws** keyword.

```

1 package propogation;
2 import java.io.FileNotFoundException;
3
4 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
5 public class ExceptionTest {
6     public static void main(String[] args)
7         throws FileNotFoundException {
8         method1();
9         System.out.println("after calling m()");
10    }
11
12    static void method1() throws FileNotFoundException{
13        method2();
14    }
15
16    static void method2() throws FileNotFoundException{
17        method3();
18    }
19
20    static void method3() throws FileNotFoundException{
21        throw new FileNotFoundException();
22    }
23
24
25 }

/*OUTPUT

Exception in thread "main" java.io.FileNotFoundException
    at propogation.ExceptionTest.method3(ExceptionTest.java:21)
    at propogation.ExceptionTest.method2(ExceptionTest.java:17)
    at propogation.ExceptionTest.method1(ExceptionTest.java:13)
    at propogation.ExceptionTest.main(ExceptionTest.java:8)

*/

```

step 1

step 2

step 3

step 4 (propagate exception)

step 5 (propagate exception)

step 6 (propagate exception)

15. Can a catch or finally block throw exception in java?

Answer. Yes, catch or finally block can throw checked or unchecked exception but it must be handled accordingly. Please refer this post for [handling checked and unchecked exceptions](#) in java.

16. Why shouldn't you use Exception for catching all exceptions in java?

Answer. Catching Exception rather than handling specific exception can be vulnerable to our application. [Multiple catch blocks](#) must be used to catch specific exceptions, because handling specific exception gives developer the liberty of taking appropriate action and develop robust application

17. What is Difference between multiple catch block and multi catch syntax?

Answer. Experienced developers must know in detail about this Exception handling interview question in java

	multiple catch block	multi catch syntax
1	multiple catch blocks were introduced in prior versions of Java 7 and does not provide any automatic resource management in java.	multi catch syntax was introduced in java 7 for improvements in multiple exception handling which helps in automatic resource management in java .
2	Here is the syntax for writing multiple catch block in java > <pre>try{ //code . . . }catch(IOException ex1){ //code . . . } catch(SQLException ex2){ //code . . . }</pre>	Here is the multi catch syntax in java > <pre>try{ //code . . . }catch(IOException SQLException ex){ //code . . . }</pre> <p>We could separate different exceptions using pipe ()</p>
3	For catching IOException and SQLException we need to write two catch block like this >	with the help of multi catch syntax we can catch IOException and SQLException in one catch block using multi catch syntax like this >

	<pre> 3@ import java.io.IOException; 4 import java.sql.SQLException; 5 6 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */ 7 public class ExceptionTest { 8@ public static void main(String[] args) { 9 10 try{ 11 int i=1; 12 if(i==1) 13 throw new IOException(); 14 else 15 throw new SQLException(); 16 }catch(SQLException ex){ 17 System.out.println(ex + " handled "); 18 }catch(IOException ex){ 19 System.out.println(ex + " handled "); 20 } 21 } 22 } /*OUTPUT 26 java.io.IOException handled 27 */ </pre>	<pre> 3@ import java.io.IOException; 4 import java.sql.SQLException; 5 6 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */ 7 public class ExceptionTest { 8@ public static void main(String[] args) { 9 10 try{ 11 int i=1; 12 if(i==1) 13 throw new IOException(); 14 else 15 throw new SQLException(); 16 }catch(IOException SQLException ex){ 17 System.out.println(ex + " handled "); 18 } 19 } 20 } /*OUTPUT 25 java.io.IOException handled 26 */ </pre>
--	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

4 When multiple catch blocks are used , first catch block could be subclass of Exception class handled in following catch blocks like this >

IOException is subclass of Exception in java.

```

3 import java.io.IOException;
4
5 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
6 public class ExceptionTest {
7@   public static void main(String[] args) {
8     try{
9       throw new IOException();
10    }catch(IOException e){
11      System.out.println("IOException handled");
12    }catch(Exception e){
13      System.out.println("Exception handled");
14    }
15  }
16 }
/*OUTPUT
20 IOException handled
21 */

```

If Multi catch syntax is used to catch subclass and its superclass than compilation error will be thrown.

IOException and Exception in multi catch syntax will cause compilation error “The exception IOException is already caught by the alternative Exception”.

```

3 import java.io.IOException;
4
5 /** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
6 public class ExceptionTest {
7@   public static void main(String[] args) {
8
9     try{
10       throw new IOException();
11    }catch(IOException | Exception ae){
12      System.out.println("Exception handled");
13    }
14  }
15 }

```

Solution >

		<p>We must use only Exception to catch its subclass like this ></p> <pre> 6 public class ExceptionTest { 7 public static void main(String[] args) { 8 9 try{ 10 throw new IOException(); 11 }catch(Exception ae){ 12 System.out.println("Exception handled"); 13 } 14 } 15 } 16 }</pre>
5	Does not provide such features.	<p>Features of multi catch syntax ></p> <ul style="list-style-type: none"> • Has improved way of catching multiple exceptions. • This syntax does not looks clumsy. • Reduces developer efforts of writing multiple catch blocks. • Allows us to catch more than one exception in one catch block. • Helps in automatic resource management.

18. can a method be overloaded on basis of exceptions in java ?

Answer.

Another Exception handling interview question which will test your practical understanding of exception in java.

Yes a method be overloaded on basis of exceptions in java.

But now question which overloaded exception will be called.

Let's take an example :

Ques. Let's say one method handles *Exception* and other handles *ArithmaticException*. Which method will be invoked when *ArithmaticException* is thrown?

Ans. Method which handles more specific exception will be called.

Program >

```

import java.io.IOException;
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com
 * Main class */
public class ExceptionTest {

    void method(Exception e){
        System.out.println(e+" caught in Exception method");
    }
    void method(ArithmaticException ae){
        System.out.println(ae+" caught in ArithmaticException method");
    }

    public static void main(String[] args) {
        ExceptionTest obj=new ExceptionTest();
        obj.method(new ArithmaticException());
        obj.method(new IOException());
    }
}
/* OUTPUT
java.lang.ArithmaticException caught in ArithmaticException method
java.io.IOException caught in Exception method
*/

```

19. What are the differences between between ClassNotFoundException and NoClassDefFoundError in java ?

Answer.

	ClassNotFoundException	NoClassDefFoundError
1	ClassNotFoundException is <u>Checked (compile time) Exception in java.</u>	1. <u>NoClassDefFoundError is a Error in java. Error and its subclasses are regarded as unchecked exceptions in java.</u>

2.	<p>3. <u>Here is the hierarchy of java.lang.ClassNotFoundException -</u></p> <ul style="list-style-type: none"> -java.lang.Object -java.lang.Throwable -java.lang.<u>Exception</u> -java.lang.ReflectiveOperationException -java.lang.ClassNotFoundException 	<p>4. <u>Here is the hierarchy of java.lang.NoClassDefFoundError -</u></p> <ul style="list-style-type: none"> -java.lang.Object -java.lang.Throwable -java.lang.<u>Error</u> -java.lang.LinkageError -java.lang.NoClassDefFoundError
3	<p>ClassNotFoundException is thrown when JVM tries to class from classpath but it does not find that class.</p>	<p>NoClassDefFoundError is thrown when JVM tries to load class which ></p> <ul style="list-style-type: none"> • was NOT available at runtime but • was available at compile time.
	<p>ExceptionInInitializerError has got nothing to do with ClassNotFoundException.</p>	<p>You must ensure that class does not throws java.lang.ExceptionInInitializerError because that is likely to be followed by NoClassDefFoundError.</p>

20. What are the most important frequently occurring Exception and Errors which you faced in java?

Answer. Most common and frequently occurring **checked (compile time)** and Errors in java >

[FileNotFoundException in java](#)

[SQLException in java](#)

[What is java.lang.InterruptedException in java](#)

[when java.lang.ClassNotFoundException occurs in java](#)

Most common and frequently occurring unchecked (**runtime**) in java.

[What is java.lang.NullPointerException in java, when it occurs, how to handle, avoid and fix it](#)

[NumberFormatException in java](#)

[IndexOutOfBoundsException in java](#)

[When java.lang.ArrayIndexOutOfBoundsException occurs in java](#)

[When java.lang.StringIndexOutOfBoundsException occurs in java](#)

[java.lang.ArithmaticException in java - Divide number by zero](#)

[When dividing by zero does not throw ArithmaticException in java](#)

[When java.lang.IllegalStateException occurs in java](#)

[when java.lang.IllegalMonitorStateException is thrown in java](#)

[Solve java.lang.UnsupportedOperationException in java](#)

Most common and frequently occurring **Errors** in java >

[OutOfMemoryError in java](#)

[When java.lang.StackOverflowError occurs in java](#)

[Solve java.lang.ExceptionInInitializerError in java](#)

[How to solve java.lang.NoClassDefFoundError in java](#)

21. What is stackTrace in exception handling?

The stack trace can be printed to the standard error by calling the `public void printStackTrace()` method of an exception.

From Java 1.4, the stack trace is encapsulated into an array of a java class called `java.lang.StackTraceElement`. The stack trace element array returned by `Throwable.getStackTrace()` method. Each element represents a single stack frame. All

stack frames except for the one at the top of the stack represent a method invocation. The frame at the top of the stack represents the execution point at which the stack trace was generated. Typically, this is the point at which the throwable corresponding to the stack trace was created.

Serialization

1. What is Serialization in java?

Let's start by understanding what is Serialization, it's most basic question which **you will have to answer almost in each and every java interview**. Serialization is process of converting **object into byte stream**.

Serialized object (byte stream) can be:

- >Transferred over network.
- >Persisted/saved into file.
- >Persisted/saved into database.

Once, object have have been transferred over network or persisted in file or in database, we could deserialize the object and retain its state as it is in which it was serialized.

2. How do we Serialize object, write a program to serialize and deSerialize object and persist it in file (Important)?

In order to serialize object our class needs to implement **java.io.Serializable** interface. Serializable interface is **Marker interface** i.e. it **does not have any methods** of its own, **but it tells Jvm that object has to converted into byte stream**.

SERIALIZATION>

Create object of ObjectOutputStream and give it's reference variable name oout and call writeObject() method and pass our employee object as parameter
`[oout.writeObject(object1)]`

```
OutputStream fout = new FileOutputStream("ser.txt");
ObjectOutput oout = new ObjectOutputStream(fout);
System.out.println("Serialization process has started, serializing employee objects...");
oout.writeObject(object1);
```

DESERIALIZATION>

Create object of ObjectInput and give it's reference variable name oin and call readObject() method [oin.readObject()]

```
InputStream fin=new FileInputStream("ser.txt");
ObjectInput oin=new ObjectInputStream(fin);
System.out.println("DeSerialization process has started, displaying employee objects... ");
Employee emp;
emp=(Employee)oin.readObject();
```

3. How can you customize Serialization and DeSerialization process when you have implemented Serializable interface (Important)?

Here comes the quite **challenging question**, where you could prove how strong your Serialization concepts are. We can customize **Serialization** process by defining **writeObject()** method & **DeSerialization** process by defining **readObject()** method.

Let's customize **Serialization** process by defining **writeObject()** method :

```
private void writeObject(ObjectOutputStream os) {
    System.out.println("In, writeObject() method.");
    try {
        os.writeInt(this.id);
        os.writeObject(this.name);
    } catch (Exception e) {
```

```

        e.printStackTrace();
    }
}

```

We have serialized id and name manually by writing them in file.

Let's customize **DeSerialization** process by defining **readObject()** method :

```

private void readObject(ObjectInputStream ois) {
    System.out.println("In, readObject() method.");
    try {
        id=ois.readInt();
        name=(String)ois.readObject();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

We have DeSerialized id and name manually by reading them from file.

4. Wap to explain how can we Serialize and DeSerialize object by implementing Externalizable interface (Important)?

For serializing object by implementing Externalizable interface, we need to override writeExternal() and readExternal() for serialization process to happen.

For **Serialization** process override **writeExternal()** method & for **DeSerialization** process by override **readExternal()** method.

Let's customize **Serialization** process by overriding [writeExternal\(\)](#) method :

```

public void writeExternal(ObjectOutput oo) throws IOException {
    System.out.println("in writeExternal()");
    oo.writeInt(id);
    oo.writeObject(name);
}

```

We have serialized id and name manually by writing them in file.

Let's customize **DeSerialization** process by overriding [**readExternal\(\)**](#) method :

```
public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
    System.out.println("in readExternal()");
    this.id=in.readInt();
    this.name=(String)in.readObject();
}
```

We have DeSerialized id and name manually by reading them from file.

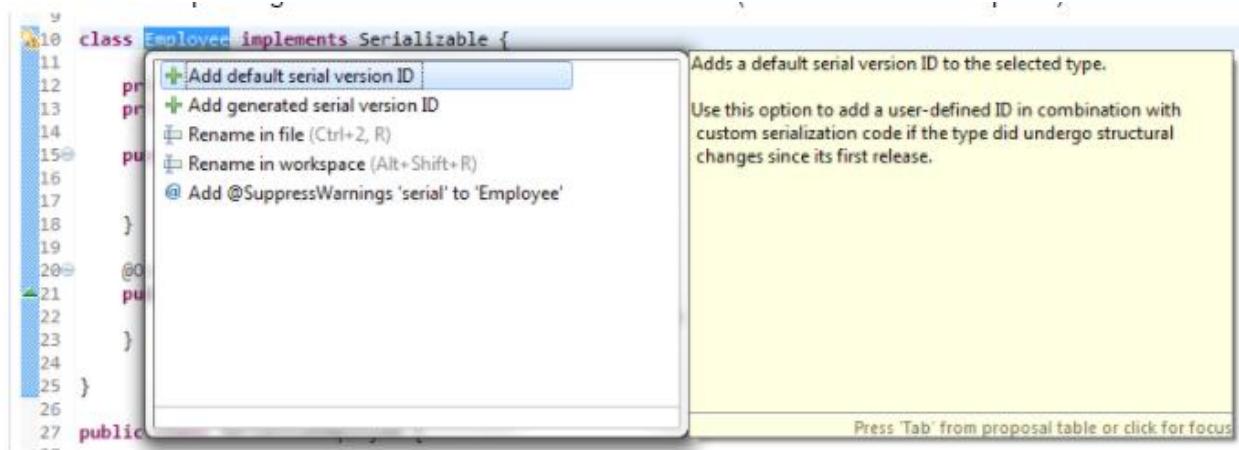
5. How can you avoid certain member variables of class from getting Serialized?

Mark member variables as **static** or **transient**, and those member variables will no more be a part of Serialization.

6. What is serialVersionUID?

Answer. The serialization at runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

We can use eclipse to generate serialVersionUID for our class (as done in below snapshot)



How to avoid **warning** 'The serializable class Employee does not declare a static final serialVersionUID field of type long' ?

Again answer is we can use eclipse to generate serialVersionUID for our class (as mentioned in above screenshot, click on warning button on left in line 10).

7. What will be impact of not defining serialVersionUID in class (Important)?

This is one my favourite question, i am going to discuss it in a very detailed manner. serialVersionUID is used for **version control of object**.

If we don't define serialVersionUID in the class, and any **modification** is made in class, then we **won't be able to deSerialize our class** because **serialVersionUID generated by java compiler for modified class will be different from old serialized object**. And deserialization process will end up throwing **java.io.InvalidClassException** (because of serialVersionUID mismatch)

Let's frame another question by twisting few words in it.

If you have serialized a class & then added few fields in it and then deserialize already serialized version of class, how can you ensure that you don't end up throwing InvalidClassException?

>Simply we need to define **serialVersionUID** in class.

When we Deserialize class (class which has been modified after Serialization and also class **doesn't declare SerialVersionUID**) **InvalidClassException** is thrown.

When we Deserialize class (class which has been modified after Serialization and also class **declare SerialVersionUID**) its gets DeSerialized **successfully**.

8. What are compatible and incompatible changes in Serialization process?

Compatible Changes : Compatible changes are those changes which **does not affect** deSerialization process even if class was updated after being serialized (provided serialVersionUID has been declared)

- **Adding new fields** - We can add new member variables in class.
- **Adding writeObject()/readObject() methods** - We may add these methods to customize serialization process.
- **Removing writeObject()/readObject() methods** - We may remove these methods and then default customization process will be used.

- **Changing access modifier of a field** - The change to access modifiers i.e. public, default, protected, and private have no effect on the ability of serialization to assign values to the fields.
- **Changing a field from static to non static OR changing transient filed to non transient field.** - it's like addition of fields.

InCompatible Changes : InCompatible changes are those changes which affect deSerialization process if class was updated after being serialized (provided serialVersionUID has been declared)

- **Deletion of fields.**
- **Changing a nonstatic field to static or non transient field to transient field.**
- it's equal to deletion of fields.
- **Modifying the writeObject() / readObject() method** - we must not modify these method, though adding or removing them completely is compatible change.

9. What if Serialization is not available, is any other alternative way to transfer object over network?

We can convert **JSON** to transfer the object. JSON is helpful in stringifying and de stringifying object.

>**Hibernate** (ORM tool) helps in persisting object as it in database and later we can read persisted object.

>We can convert object into **XML** (as done in web services) and transfer object over network.

10. Why static member variables are not part of java serialization process (Important)?

Answer. Serialization is applicable on objects or primitive data types only, but **static** members are **class level variables**, therefore, **different object's of same class have same value for static member.**

So, serializing static member will consume unnecessary space and time.

Also, if modification is made in static member by any of the object, it won't be in sync with other serialized object's value.

11. What is significance of transient variables?

Answer. Serialization is not applicable on transient variables (it helps in saving time and space during Serialization process), we **must mark all rarely used variables as transient**. We can initialize transient variables during deSerialization by customizing deSerialization process.

12. What will happen if one the member of class does not implement Serializable interface (Important)?

This is classy question which will check your in depth knowledge of Serialization concepts. If any of the member does not implement Serializable than **NotSerializableException** is thrown

13. What will happen if we have used List, Set and Map as member of class?

Answer. This question which will check your in depth knowledge of Serialization and Java Api's. ArrayList, HashSet and HashMap implements Serializable interface, so if we will use them as member of class they will get Serialized and DeSerialized as well.

14. Is constructor of class called during DeSerialization process?

Answer. This question which will check your in depth knowledge of Serialization and constructor chaining concepts. It depends on whether our object has implemented Serializable or Externalizable.

If **Serializable** has been implemented - constructor is **not called** during DeSerialization process.

But, if **Externalizable** has been implemented - constructor is **called** during DeSerialization process.

15. Are primitive types part of serialization process?

Answer. Yes, [primitive types are part of serialization process](#). Interviewer tends to check your basic java concepts over here.

16. What values will int and Integer will be initialized to during DeSerialization process if they were not part of Serialization?

[int will be initialized to 0 and Integer will be initialized to null during DeSerialization](#) (if they were not part of Serialization process).

17. What is singleton?

In Java ,we can create objects by calling constructor.But imagine a scenario where we want to control object instantiation.There could be many reasons why we want to control the object creation.

Normally in 3 tier architecture we create single instance of service and DAO objects since we don't want to create multiple DAO objects as number of database connections are limited and by creating multiple DAO objects we do not want to exhaust database connections.

This is just one example ,there could be multiple such examples in real world.

Code snippet for a singleton class

Here I am using double check mechanism for creating a singleton instance.

```
package com.kunaal.algo;

import java.io.Serializable;

/**
 * Here we are making ConnectionFactory as a singleton.
 * Since we want connection factory to be initiated once and used
 * by different classes of the project.
 *
 * We also want to read the connection parameters once and use it as
 * a place holder for pooled connections.
 *
 * @author KunaalATrehan
 *
 */
public class ConnectionFactory implements Serializable{
    //Static variable for holding singleton reference object
    private static ConnectionFactory INSTANCE;

    /**
     * Private constructor
     */
    private ConnectionFactory() {
    }

    /**
     * Static method for fetching the instance
     * @return
     */
    public static ConnectionFactory getInstance(){
        //Check whether instance is null or not
        if(INSTANCE ==null){
            //Locking the class object
            synchronized(ConnectionFactory.class) {
                //Doing double check for the instance
                //This is required in case first time two threads simultaneously invoke
                //getInstance().So when another thread get the lock,it should not create
                //the
                //object again as its already created by the previous thread.
                if(INSTANCE==null)
                    INSTANCE=new ConnectionFactory();
            }
        }
        return INSTANCE;
    }
}
```

```

        }
    }

    return INSTANCE;
}
}
}

```

18. What happens when we serialize the singleton?

Serialization allows storing the object in some data store and re create it later on. However when we serialize a singleton class and invoke deserialization multiple times. We can end up with multiple objects of the singleton class. Even though constructor is private, deserialization process gets hold of the private constructor while recreating the object from the serialized data store.

So can we avoid it. **Yes we can avoid it.** We will go through step by step and explain what needs to be done when we reconstruct the object from the serialized data store so that singleton behavior is not broken when object reconstruction happens.

Case-1: Serialization breaking singleton behavior

Here we are serializing the singleton instance and reading it multiple times. So we will see that INSTANCE reference is same, however multiple objects are created.

```

package com.kunaal.algo;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

/**
 * @author KunaalATrehan
 *
 */
public class SerializationTest {
    /**
     * @param args
     * @throws IOException
     * @throws FileNotFoundException
     * @throws ClassNotFoundException
     */

```

```

public static void main(String[] args) throws FileNotFoundException,
IOException, ClassNotFoundException {
    ConnectionFactory INSTANCE=ConnectionFactory.getInstance();

    //Here I am serializing the connection factory instance
    ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("connFactory.ser"));
    oos.writeObject(INSTANCE);
    oos.close();

    //Here I am recreating the instance by reading the serialized object
data store
    ObjectInputStream ois = new ObjectInputStream(new
FileInputStream("connFactory.ser"));
    ConnectionFactory factory1 = (ConnectionFactory) ois.readObject();
    ois.close();

    //I am recreating the instance AGAIN by reading the serialized object
data store
    ObjectInputStream ois2 = new ObjectInputStream(new
FileInputStream("connFactory.ser"));
    ConnectionFactory factory2 = (ConnectionFactory) ois2.readObject();
    ois2.close();

    //Lets see how we have broken the singleton behavior

    System.out.println("Instance reference check->" +
factory1.getInstance());
    System.out.println("Instance reference check->" +
factory2.getInstance());

System.out.println("=====");
System.out.println("Object reference check->" +factory1);
System.out.println("Object reference check->" +factory2);
}
}

```

Output is as follows:-

```

Instance reference check->com.kunaal.algo.ConnectionFactory@763f5d
Instance reference check->com.kunaal.algo.ConnectionFactory@763f5d

```

```
Object reference check->com.kunaal.algo.ConnectionFactory@13a317a
Object reference check->com.kunaal.algo.ConnectionFactory@186768e
```

Case-2: Serialization and singleton working properly

In order to make serialization and singleton work properly, we have to introduce `readResolve()` method in the singleton class. `readResolve()` method lets developer control what object should be returned on deserialization.

For the current `ConnectionFactory` singleton class, `readResolve()` method will look like this.

```
/*
 * Special hook provided by serialization where developer can control what
object needs to sent.
 * However this method is invoked on the new object instance created by de
serialization process.
 * @return
 * @throws ObjectStreamException
 */
private Object readResolve() throws ObjectStreamException{
    return INSTANCE;
}
```

Output is as follows:-

```
Instance reference check->com.kunaal.algo.ConnectionFactory@13a317a
Instance reference check->com.kunaal.algo.ConnectionFactory@13a317a
=====
Object reference check->com.kunaal.algo.ConnectionFactory@13a317a
Object reference check->com.kunaal.algo.ConnectionFactory@13a317a
```

So now serialization and singleton is working properly and it does not matter how many times we read the serialized format of singleton object. We will get the one instance. `readResolve()` did the trick

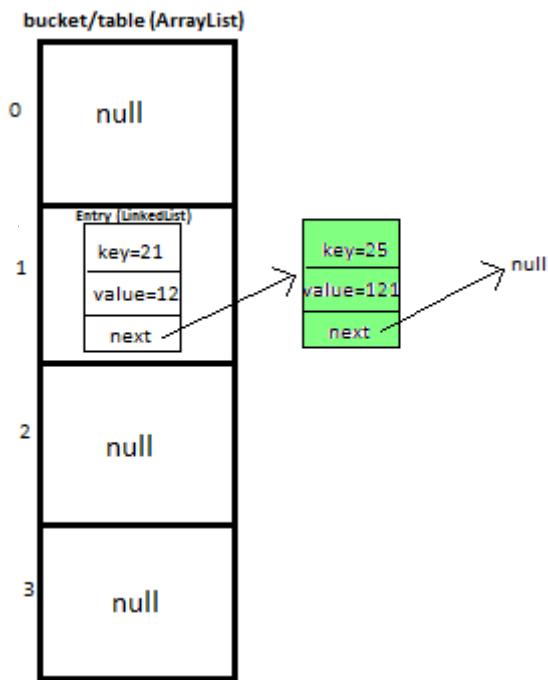
Collections

1. Can we use custom object as key in HashMap? If yes then how?

For using object as Key in HashMap, we must implements [equals and hashCode method](#).

2. Why do we need to override equals and hashCode method?

Before understanding the concept of overriding `equals()` and `hashCode()` method, we must understand what is **bucket, Entry, and Entry.next**



Bucket is [ArrayList](#) of **Entry**.

Entry is [LinkedList](#) which contains information about key, value and next.

Entry.next points to next Entry in [LinkedList](#).

3. Why to override hashCode method?

It helps in finding bucket location, where entry(with key-value pair) will be stored .

Entry (of type [LinkedList](#)) is stored in **bucket (ArrayList)**.

If, **hashCode() method is overridden properly**, we will find bucket location using **hashCode()** method, we will obtain **Entry** on that bucket location, then iterate over each and every **Entry** (by calling **Entry.next**) and check whether new and existing keys are equal or not. If keys are equal replace key-value in **Entry** with new one, else call **Entry.next**. But, now the question comes how to check whether two keys are equal or not. So, it's time to implement **equals()** method.

If, **hashCode method is not overridden** for same key every time **hashCode()** method is called it might produce different hashcode, there might happen **2 cases** i.e. when **put** and **get** method are called.

Case 1 : when put() method is called-

There might be possibility that same **Entry** (with key-value pair) will get stored at multiple locations in **bucket**.

Conclusion-> key- value pair may get stored multiple times in **HashMap**.

Case 2 : when get() method is called-

As there is possibility that **hashCode()** method might return different hashcode & rather than searching on bucket location where **Entry**(with key) exists we might be searching for key on some other bucket location.

Conclusion> key existed in HashMap, but still we were not able to locate the bucket location in which it was stored.

4. Why to override equals method?

Once we have located bucket location in which our Entry (with key-value pair) will be stored, **Equals** method helps us in finding whether **new and existing keys are equal or not.**

If we equals method is not overridden - though we will be able to find out correct bucket location if hashCode() method is overridden correctly, but still if equals method is not overridden, there might happen **2 cases** i.e. when **put and get method** are called.

Case 1 : when put() method is called-

we might end up storing new Entry (with new key-value pair) multiple times on same bucket location (because of absence of equals method, we don't have any way of comparing key's),

In this case, even if keys are equal, we will keep on calling Entry.next until we reach last Entry on that bucket location and ultimately we will end up storing new Entry (with new key) again in same bucket location.

Conclusion> key- value pair stored multiple times in HashMap.

Case 2 : when get() method is called-

we won't be able to compare two keys (new key with existing **Entry.key**) and we will call **Entry.next** and again we won't be able to compare two keys and ultimately when **Entry.next** is null - we will return **false**.

Conclusion> key existed in HashMap, but still we were not able to retrieve it.

So, it's important to override equals method to check equality of two keys.

5. If two objects have same hashcode, are they always equal?

No, It's not necessary that object's having same hashcode are always equal. Because same hashcode means object are stored on same bucket location, as key/object in bucket is stored in Entry(**Linked List**), key/object's might be stored on Entry.next (i.e. on some different entry)

6. If two objects equals() method return true, do objects always have same hashcode?

Yes, two objects can return true only if they are stored on same bucket location.

First, hashCode() method must have returned same hashCode for both objects, than on that bucket location's Entry key.equals() is called, which returns true to confirm objects/keys are equal.

So, if object's equals return true, they always have same hashCode.

7. What classes should i prefer to use a key in HashMap?

This question will check your in depth knowledge of Java's Collection API's. We should prefer **String, Integer, Long, Double, Float, Short and any other wrapper class**. Reason behind using them as a key is that they override equals() and hashCode() method, we need not to write any explicit code for overriding equals() and hashCode() method.

Let's use Integer class as key in HashMap.

```
import java.util.HashMap;
import java.util.Map;
public class StringInMap {
    public static void main(String...a){

        //HashMap's key=Integer class (Integer's api has already overridden
        //hashCode() and equals() method for us )
        Map<Integer, String> hm=new HashMap<Integer, String>();
        hm.put(1, "data");
        hm.put(1, "data OVERRIDDEN");

        System.out.println(hm.get(1));

    }
}
/*OUTPUT
data OVERRIDDEN
*/
```

If, we note above program, what we will see is we didn't override equals() and hashCode() method, but still we were able to store data in HashMap, override data and retrieve data using get method.

>Let's check in **Integer's API**, how Integer class has overridden equals() and hashCode() method :

```
public int hashCode() {
```

```

        return value;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Integer) {
            return value == ((Integer)obj).intValue();
        }
        return false;
    }
}

```

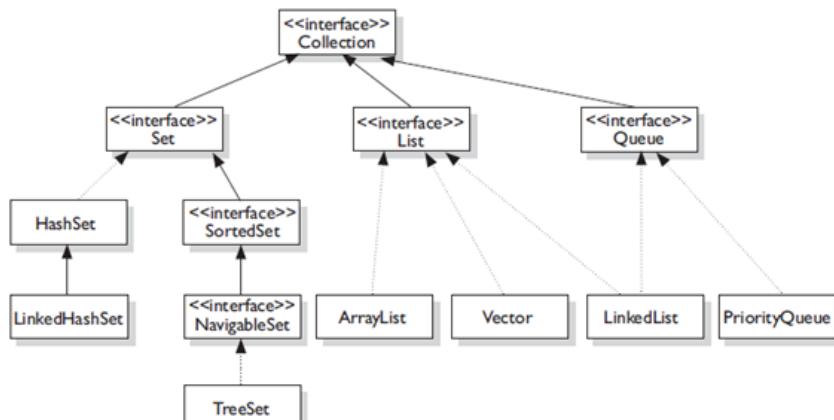
8. Can overriding of hashCode() method cause any performance issues?

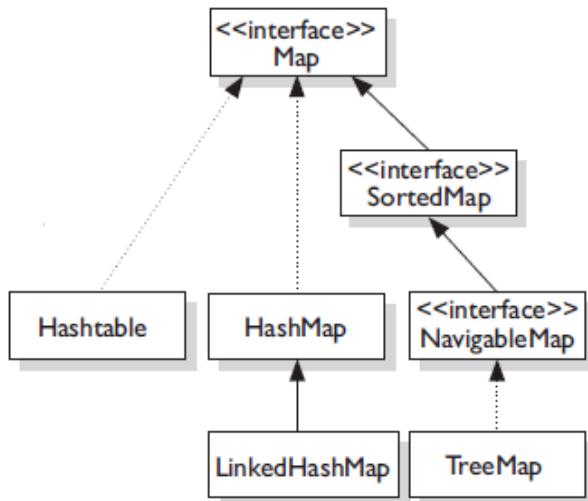
Improper implementation of hashCode() can cause performance issues, because in that most of the key-value pairs will be stored on same bucket location and unnecessary time will be consumed while fetching value corresponding to key.

9. What are subinterfaces of Collection interface in java? Is Map interface also a subinterface of Collection interface in java?

List and Set are subinterfaces of java.util.Collection in java.

It's important to note Map interface is a member of the Java Collections Framework, but it does not implement Collection interface in java.





10. What are differences between ArrayList and LinkedList in java?

Answer. This is very important collection framework interview question in java.

	Property	<code>java.util.ArrayList</code>	<code>java.util.LinkedList</code>																				
1	Structure	<p>java.util.ArrayList is index based structure in java.</p> <table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td> </tr> <tr> <td>71</td><td>null</td><td>null</td><td>null</td><td>null</td><td>null</td><td>null</td><td>null</td><td>null</td><td>null</td> </tr> </table>	0	1	2	3	4	5	6	7	8	9	71	null	<p>A <code>java.util.LinkedList</code> is a data structure consisting of a group of nodes which together represent a sequence.</p> <p>node is composed of a data and a reference (in other words, a link) to the next node in the sequence in java.</p>								
0	1	2	3	4	5	6	7	8	9														
71	null	null	null	null	null	null	null	null	null														

2	Resizable	ArrayList is Resizable-array in java.	New node is created for storing new element in LinkedList in java.
3	Initial capacity	java.util.ArrayList is created with initial capacity of 10 in java.	For storing every element node is created in LinkedList, so linkedList's initial capacity is 0 in java.
4	Ensuring Capacity/ resizing.	ArrayList is created with initial capacity of 10. ArrayList's size is increased by 50% i.e. after resizing it's size become 15 in java.	For storing every element node is created, so linkedList's initial capacity is 0, it's size grow with addition of each and every element in java.
5	RandomAccess interface	ArrayList implements RandomAccess(Marker interface) to indicate that they support fast random access (i.e. index based access) in java.	LinkedList does not implement RandomAccess interface in java.
6	AbstractList and AbstractSequentialList	ArrayList extends AbstractList (abstract class) which provides implementation to List interface to minimize the effort required to implement this interface backed by RandomAccess interface.	LinkedList extends AbstractSequentialList (abstract class), AbstractSequentialList extends AbstractList. In LinkedList, data is accessed sequentially, so for obtaining data at specific index, iteration is done on nodes sequentially in java.
7	How get(index) method works? (Though difference has been discussed briefly in above	Get method of ArrayList directly gets element on specified index. Hence, offering O(1) complexity in java.	Get method of LinkedList iterates on nodes sequentially to get element on specified index. Hence, offering O(n) complexity in java.

	2 points but in this in point we will figure difference in detail.)		
8	When to use	Use ArrayList when get operations is more frequent than add and remove operations in java.	Use LinkedList when add and remove operations are more frequent than get operations in java.

11. What are differences between List and Set interface in java?

Answer. Another very very important collection framework interview question to differentiate between **List** and **Set** in java.

	Property	<i>java.util.List</i>	<i>java.util.Set</i>
1	Insertion order	java.util.List is ordered collection it maintain insertion order in java.	<p><i>Most of the java.util.Set implementation</i> does not maintain insertion order.</p> <p>HashSet does not maintains insertion order in java.</p> <p>Thought LinkedHashSet maintains insertion order in java.</p> <p>TreeSet is sorted by natural order in java.</p>

2	Duplicate elements	List allows to store duplicate elements in java.	Set does not allow to store duplicate elements in java.
3	Null keys	List allows to store many null keys in java.	Most of the Set implementations allow to add only one null in java. TreeSet does not allow to add null in java.
4	Getting element on specific index	<p>List implementations provide get method to get element on specific index in java.</p> <p>ArrayList, Vector, copyOnWriteArrayList and LinkedList provides -</p> <p><i>get(int index)</i></p> <p>Method returns element on specified <i>index</i>.</p> <p>Get method directly gets element on specified index. Hence, offering O(1) complexity.</p>	Set implementations does not provide any such get method to get element on specified index in java.
5	Implementing classes	ArrayList , LinkedList , Vector , CopyOnWriteArrayList classes implements List interface in java.	HashSet , CopyOnWriteArraySet , LinkedHashSet , TreeSet , ConcurrentSkipListSet , EnumSet classes implements Set interface in java.
6	listIterator	listIterator method returns listIterator to iterate over elements in List in java.	Set does not provide anything like listIterator. It simply return Iterator in java.

		<p>listIterator provides additional methods as compared to iterator like</p> <p>hasPrevious(), previous(), nextIndex(), previousIndex(), add(E element), set(E element)</p>	
7	Structure and resizable	<p>List are Resizable-array implementation of the <code>java.util.List</code> interface in java.</p>	<p>Set uses Map for their implementation.</p> <p>Hence, structure is map based and resizing depends on Map implementation.</p> <p><i>Example > HashSet internally uses HashMap.</i></p>
8	Index based structure /RandomAccess	<p>As ArrayList uses array for implementation it is index based structure, hence provides random access to elements.</p> <p>But LinkedList is not indexed based structure in java.</p>	<p>Set is not index based structure at all in java.</p>

12. What are differences between Iterator and ListIterator? in java

Answer. This collection framework interview question is tests your knowledge of iterating over different collection framework classes in java.

	<i>java.util.ListIterator</i>	<i>java.util.Iterator</i>
--	--------------------------------------	----------------------------------

1	<p>hasPrevious() method returns true if this listIterator has more elements when traversing the list in the reverse direction.</p>	No such method in java.util.Iterator.
2	<p>previous() returns previous element in iteration (traversing in backward direction).</p> <p>if the iteration has no previous elements than NoSuchElementException is thrown.</p>	No such method in java.util.Iterator.
3	<p>nextIndex() method returns the index of the element that would be returned by a subsequent call to next() method. If listIterator is at the end of the list than method returns size of list.</p>	No such method in java.util.Iterator.
4	<p>previousIndex() method returns the index of the element that would be returned by a subsequent call to previous() method. If listIterator is at the start of the list than method returns -1.</p>	No such method in java.util.Iterator.
5	<p>add(E element)</p> <p>Method inserts the specified element into the list.</p> <p>The element is inserted immediately before the element that would be returned by next (<i>So, subsequent call to next would be unaffected</i>), if any, and after the element that would be returned by previous (<i>So, subsequent call to previous would return the new element</i>), if any.</p> <p>If the list does not contain any element than new element will be the sole element in the list.</p>	No such method in java.util.Iterator.
6	<p>set(E element)</p> <p>Method replaces the last element returned by next() or previous() method with the specified element. This call can be made only if neither remove nor add have been called after the last call to next or previous.</p> <p>If call to set() method is followed up by any call made to remove() or add() method after next() or previous() than UnsupportedOperationException is thrown.</p>	No such method in java.util.Iterator.

7	All the implementations of List interface like ArrayList , LinkedList , Vector , CopyOnWriteArrayList classes returns listIterator.	All Implementation classes of Collection interface's subinterfaces like Set and List return iterator.
---	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------

13. What are differences between Collection and Collections in java?

`java.util.Collection` is the root **interface** in the *hierarchy of Java Collection framework*.

The JDK does not provide any classes which directly implements `java.util.Collection` interface, but it provides classes such as [ArrayList](#), [LinkedList](#), [vector](#), [HashSet](#), [EnumSet](#), [LinkedHashSet](#), [TreeSet](#), [CopyOnWriteArrayList](#), [CopyOnWriteArraySet](#), [ConcurrentSkipListSet](#) which implements more specific subinterfaces like [Set](#) and [List](#) in java.

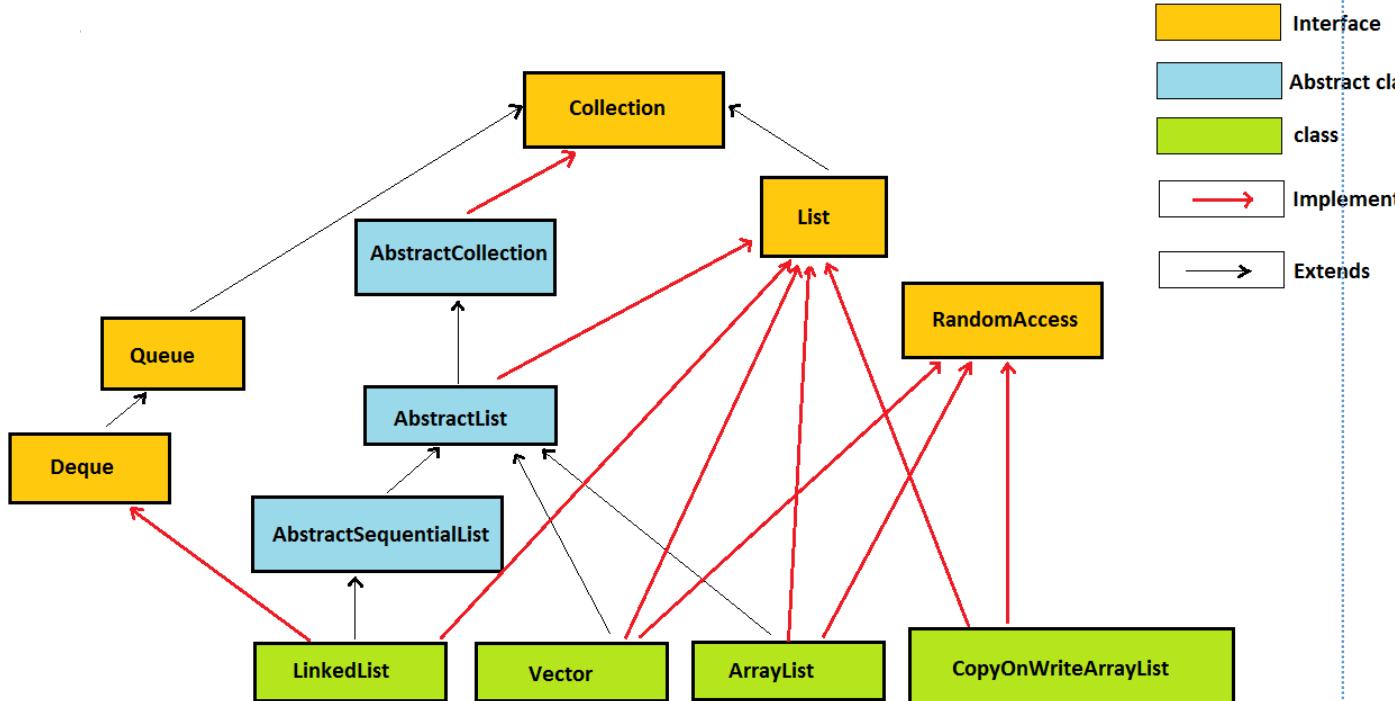
`java.util.Collections` is a utility **class** which **consists of static methods** that **operate on or return Collection** in java.

`java.util.Collections` provides method like >

- **reverse** method for reversing [List](#) in java.
- **shuffle** method for shuffling elements of [List](#) in java.
- **unmodifiableCollection**, [unmodifiableSet](#), [unmodifiableList](#), [unmodifiableMap](#) methods for making [List](#), [Set](#) and [Map](#) unmodifiable in java.
- **min** method to return smallest element in [Collection](#) in java.
- **max** method to return largest element in [Collection](#).
- **sort** method for sorting [List](#).
- **synchronizedCollection**, [synchronizedSet](#), [synchronizedList](#), [synchronizedMap](#) methods for synchronizing [List](#), [Set](#) and [Map](#) respectively in java

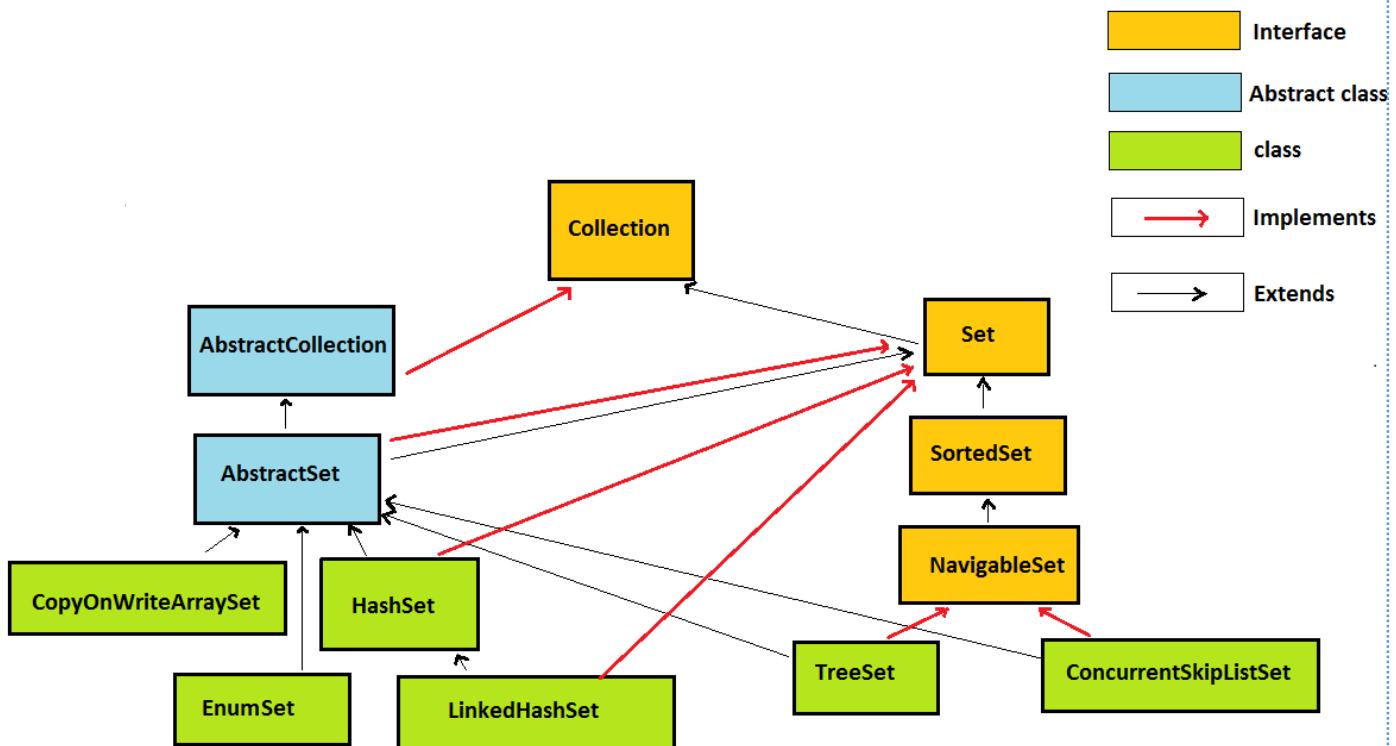
14. What are core classes and interfaces in java.util.List hierarchy in java?

Answer. Freshers must know core classes in List hierarchy but experienced developers must be able to explain this java.util.List hierarchy in detail.



15. What are core classes and interfaces in java.util.Set hierarchy?

Answer. Freshers must know core classes in Set hierarchy but experienced developers must be able to explain this java.util.Set hierarchy in detail.



16. What are differences between Iterator and Enumeration in java?

Answer. Experienced developers must be well versed to answer this collection framework interview question in java.

	Property	java.util.Enumeration	java.util.Iterator
1	Remove elements during iteration	java.util.Enumeration doesn't allows to remove elements from collection during iteration in java.	java.util.Iterator allows to remove elements from collection during iteration by using remove() method in java.
2	Improved naming conventions in Iterator	nextElement() Method Returns the next element of this enumeration if this enumeration object has at least one more element to provide.	nextElement() has been changed to next() in Iterator. And

		hasMoreElements() returns true if enumeration contains more elements.	hasMoreElements() has been changed to hasNext() in Iterator.
3	Introduced in which java version	Enumeration was introduced in first version of java i.e. JDK 1.0	Iterator was introduced in second version of java i.e. JDK 2.0 Iterator was introduced to replace Enumeration in the Java Collections Framework.
4	Recommendation	Java docs recommends iterator over enumeration.	Java docs recommends iterator over enumeration.
5	Enumeration and Iterator over Vector	Enumeration returned by Vector is fail-safe , means any modification made to Vector during iteration using Enumeration don't throw any exception in java.	Iterator returned by Vector are fail-fast , means any structural modification made to ArrayList during iteration will throw ConcurrentModificationException in java.

17. What are differences between HashMap and Hashtable in java?

Answer. Fresher and Experienced developers must answer this important collection framework interview question in detail in java.

[**Differences between java.util.*HashMap* and java.util.*Hashtable* in java >**](#)

	Property	java.util.HashMap	java.util.Hashtable
1	synchronization	java.util.HashMap is not synchronized (because 2 threads on same HashMap)	java.util.Hashtable is synchronized (because 2 threads on same Hashtable)

		object can access it at same time) in java.	object cannot access it at same time) in java.
2	Performance	<p>HashMap is not synchronized, hence its operations are faster as compared to Hashtable in java.</p>	<p>Hashtable is synchronized, hence its operations are slower as compared to HashMap in java.</p> <p>If we are working not working in multithreading environment jdk recommends us to use HashMap.</p>
3	Null keys and values	<p>HashMap allows to store one null key and many null values i.e. many keys can have null value in java.</p>	<p>Hashtable does not allow to store null key or null value.</p> <p>Any attempt to store null key or value throws runtimeException (NullPointerException) in java.</p>
4	Introduced in which java version	<p>HashMap was introduced in second version of java i.e. JDK 2.0</p>	<p>Hashtable was introduced in first version of java i.e. JDK 1.0</p> <p>But it was refactored in java 2 i.e. JDK 1.2 to implement the Map interface, hence making it a member of member of the Java Collections Framework.</p>
5	Recommendation	In non-multithreading environment it is recommended to use	<p>In java 5 i.e. JDK 1.5, it is recommended to use ConcurrentHashMap than using Hashtable.</p>

		HashMap than using Hashtable in java.	
6	Extends Dictionary (Abstract class, which is obsolete)	HashMap does not extends Dictionary in java.	Hashtable extends Dictionary (which maps non-null keys to values. In a given Dictionary we can look up value corresponding to key) in java.

18. when to use HashSet vs LinkedHashSet vs TreeSet in java?

	Property	<i>java.util.HashSet</i>	<i>java.util.LinkedHashSet</i>	<i>java.util.TreeSet</i>
1	Insertion order	java.util.HashSet does not maintains insertion order in java. Example in java > <code>set.add("b");</code> <code>set.add("c");</code> <code>set.add("b");</code> <code>set.add("c");</code> <code>set.add("a");</code> Output > No specific order	java.util.LinkedHashSet maintains insertion order in java. Example in java > <code>set.add("b");</code> <code>set.add("c");</code> <code>set.add("a");</code> Output > b c a	java.util.TreeSet is sorted by natural order in java. Example in java > <code>set.add("b");</code> <code>set.add("c");</code> <code>set.add("a");</code> Output > a b c
2	Null elements	HashSet allows to store one null in java.	LinkedHashSet allows to store one null in java.	TreeSet does not allows to store any null in java.

				Any attempt to add null throws runtimeException (NullPointerException) .
3	Data structure internally used for storing data	For storing elements HashSet internally uses HashMap.	For storing elements LinkedHashSet internally uses LinkedHashMap.	For storing elements TreeSet internally uses TreeMap.
4	Introduced in which java version	java.util.HashSet was introduced in second version of java (1.2) i.e. JDK 2.0	java.util.LinkedHashSet was introduced in second version of java (1.4) i.e. JDK 4.0	java.util.TreeSet was introduced in second version of java (1.2) i.e. JDK 2.0
5	Implements which; interface	HashSet implements java.util.Set interface.	LinkedHashSet implements java.util.Set interface.	TreeSet implements java.util.Set java.util.SortedSet java.util.NavigableSet interface.

19. What are differences between HashMap and ConcurrentHashMap in java?

Answer. Take my words java developers won't be able to get away from this very important collection framework interview question.

Differences between [java.util.HashMap](#) and [java.util.concurrent.ConcurrentHashMap](#) in java >

Property	java.util.HashMap	java.util.concurrent.ConcurrentHashMap
----------	-----------------------------------	--------------------------------------------------------

synchronization	HashMap is not synchronized.	ConcurrentHashMap is synchronized.
2 threads on same Map object can access it at concurrently?	Yes, because HashMap is not synchronized.	<p>Yes.</p> <p>But how despite of being synchronized, 2 threads on same <i>ConcurrentHashMap</i> object can access it at same time?</p> <p><i>ConcurrentHashMap</i> is divided into different segments based on concurrency level. So different threads can access different segments concurrently.</p>
Performance	<p>We will synchronize HashMap and then compare its performance with ConcurrentHashMap.</p> <p><i>We can synchronize hashMap by using Collections's class synchronizedMap method.</i></p> <div style="border: 1px solid black; padding: 5px;"> <pre>Map synchronizedMap = Collections.synchronizedMap(map);</pre> </div> <p><i>Now, no 2 threads can access same instance of map concurrently.</i></p> <p>Hence synchronized HashMap's performance is slower as compared to ConcurrentHashMap.</p>	<p>ConcurrentHashMap's performance is faster as compared to HashMap (because it is divided into segments, as discussed in above point).</p> <p><i>Read this post for performance comparison between HashMap and ConcurrentHashMap.</i></p>

	<p>But why we didn't compared HashMap (unSynchronized) with ConcurrentHashMap? Because performance of unSynchronized collection is always better than some synchronized collection. As, default (unSynchronized) hashMap didn't cause any locking.</p>	
Null keys and values	<p>HashMap allows to store one null key and many null values i.e. any key can have null value.</p>	<p>ConcurrentHashMap does not allow to store null key or null value. Any attempt to store null key or value throws runtimeException (NullPointerException).</p>
Iterators	<p>The iterators returned by the iterator() method of HashMap are <u>fail-fast</u> ></p> <p><code>hashMap.keySet().iterator()</code> <code>hashMap.values().iterator()</code> <code>hashMap.entrySet().iterator()</code></p> <p>all three iterators are fail-fast</p>	<p>iterators are <u>fail-safe</u>.</p> <p><code>concurrentHashMap.keySet().iterator()</code> <code>concurrentHashMap.values().iterator()</code> <code>concurrentHashMap.entrySet().iterator()</code></p> <p>all three iterators are fail-safe.</p>
putIfAbsent	HashMap does not contain putIfAbsent method.	If map does not contain specified key , put specified key-value pair in map and return null.

	<p><i>putIfAbsent</i> method is equivalent to writing following code ></p> <pre>synchronized (map){ if (!map.containsKey(key)) return map.put(key, value); else return map.get(key); }</pre> <p><u>Program to create method that provides functionality similar to putIfAbsent method of ConcurrentHashMap and to be used with HashMap</u></p>	If map already contains specified key , return value corresponding to specified key .
Introduced in which java version	HashMap was introduced in java 2 i.e. JDK 1.2 ,	ConcurrentHashMap was introduced in java 5 i.e. JDK 1.5 , since its introduction Hashtable has become obsolete, because of concurrency level its performance is better than Hashtable.
Implements which interface	HashMap implements java.util.Map	ConcurrentHashMap implements java.util.Map and java.util.concurrent.ConcurrentMap
Package	HashMap is in java.util package	ConcurrentHashMap is in java.util.concurrent package.

20. When to use HashMap vs Hashtable vs LinkedHashMap vs TreeMap in java?

Answer. Another important collection framework interview question to differentiate between **following Map implementations** in java.

Differences between java.util.[HashMap](#) vs java.util.[Hashtable](#) vs java.util.[LinkedHashMap](#) vs java.util.[TreeMap](#) >

Property	HashMap	Hashtable	LinkedHashMap	TreeMap
1 Insertion order	HashMap does not maintains insertion order in java.	Hashtable does not maintains insertion order in java.	LinkedHashMap maintains insertion order in java.	TreeMap is sorted by natural order of keys in java.
2 Performance	HashMap is not synchronized, hence its operations are faster as compared to Hashtable.	Hashtable is synchronized, hence its operations are slower as compared HashMap. If we are working not working in multithreading environment jdk recommends	LinkedHashMap must be used only when we want to maintain insertion order. Time and space overhead is there because for maintaining order it internally uses Doubly Linked list .	TreeMap must be used only when we want sorting based on natural order. Otherwise sorting operations cost performance. (Comparator is called for sorting purpose)

			us to use HashMap.		
3	Null keys and values	HashMap allows to store one null key and many null values i.e. many keys can have null value in java.	Hashtable does not allow to store null key or null value. Any attempt to store null key or value throws runtimeException (NullPointerException) in java.	LinkedHashMap allows to store one null key and many null values i.e. any key can have null value in java.	TreeMap does not allow to store null key but allow many null values. Any attempt to store null key throws runtimeException (NullPointerException) in java.
4	Implements which interface	HashMap implements java.util.Map	Hashtable implements java.util.Map	LinkedHashMap implements java.util.Map	TreeMap implements java.util.Map java.util.Sorted Map java.util.NavigableMap
5	Implementation uses?	HashMap use buckets	Hashtable use buckets	LinkedHashMap uses doubly linked lists	TreeMap uses Red black tree
6	Complexity of put, get and remove methods	O(1)	O(1)	O(1) overhead of updating Doubly Linked list for maintaining order it internally uses.	O(log(n))

7	Extends java.util.Dictionary (Abstract class, which is obsolete)	HashMap doesn't extends Dictionary.	Hashtable extends Dictionary (which maps non-null keys to values. In a given Dictionary we can look up value corresponding to key)	LinkedHashMap doesn't extends Dictionary.	TreeMap doesn't extends Dictionary.
8	Introduced in which java version?	HashMap was introduced in second version of java i.e. JDK 2.0	Hashtable was introduced in first version of java i.e. JDK 1.0 But it was refactored in java 2 i.e. JDK 1.2 to implement the Map interface, hence making it a member of member of the Java Collections Framework .	LinkedHashMap was introduced in fourth version of java i.e. JDK 4.0	TreeMap was introduced in second version of java i.e. JDK 2.0

21. What are differences between HashMap vs IdentityHashMap in java?

Answer. This is tricky and complex collection framework interview question for experienced developers in java.

[**Differences between java.util.HashMap and java.util.IdentityHashMap in java >**](#)

	Property	<i>java.util.HashMap</i>	<i>java.util.IdentityHashMap</i>
1	Keys comparison object-equality vs reference-equality	HashMap when comparing keys (and values) performs object-equality not reference-equality. In an HashMap, two keys k1 and k2 are equal if and only if (k1==null ? k2==null : k1.equals(k2))	IdentityHashMap when comparing keys (and values) performs reference-equality in place of object-equality. In an IdentityHashMap, two keys k1 and k2 are equal if and only if (k1==k2)
2	Initial size	Constructs a new HashMap, Its initial capacity is 16 in java. <code>new HashMap();</code>	Constructs a new IdentityHashMap, with maximum size of 21 in java. <code>new IdentityHashMap();</code>
3	Introduced in which java version	HashMap was introduced in second version of java i.e. JDK 2.0	IdentityHashMap was introduced in fourth version of java i.e. JDK 4.0
4	<i>Program</i>	Program 1 shows > <i>comparing keys (and values) performs object-equality in place of reference-equality . In an HashMap, two keys k1 and k2 are equal if and only if (k1==null ? k2==null : k1.equals(k2)).</i>	Program 2 shows > <i>comparing keys (and values) performs reference-equality in place of object-equality. In an IdentityHashMap, two keys k1 and k2 are equal if and only if (k1==k2).</i>
5	overridden equals() and hashCode() method call?	<u>overridden equals() and hashCode() method</u> are called when put, get methods are called in HashMap .	<u>overridden equals() and hashCode() method</u> are not called when put, get methods are called in IdentityHashMap .

		As shown in Program 3.	<i>Because IdentityHashMap implements equals() and hashCode() method by itself and checks for reference-equality of keys.</i> As shown in Program 4.
6	Application - can maintain <i>proxy object</i>	HashMap cannot be used to maintain <i>proxy object</i> .	IdentityHashMap can be used to maintain <i>proxy objects</i> . For example, we might need to maintain proxy object for each object debugged in the program.

22. What is WeakHashMap in java?

Answer. Another tricky collection framework interview question for experienced developers in java.

java.util.[WeakHashMap](#) is hash table based implementation of the Map interface, with *weak keys*.

An entry in a WeakHashMap will be automatically removed by garbage collector when its key is no longer in ordinary use. Mapping for a given key will not prevent the key from being discarded by the garbage collector, (i.e. made finalizable, finalized, and then reclaimed). When a key has been discarded its entry is removed from the map in java.

23. What is the difference between fail fast and fail safe iterators?

Fail-Fast Iterators

Fail-Fast iterators doesn't allow modifications of a collection while iterating over it.

Fail-Safe Iterators

Fail-Safe iterators allow modifications of a collection while iterating over it.

These iterators

throw *ConcurrentModificationException* if a collection is modified while iterating over it.

They use original collection to traverse over the elements of the collection.

These iterators don't require extra memory.

Ex : Iterators returned

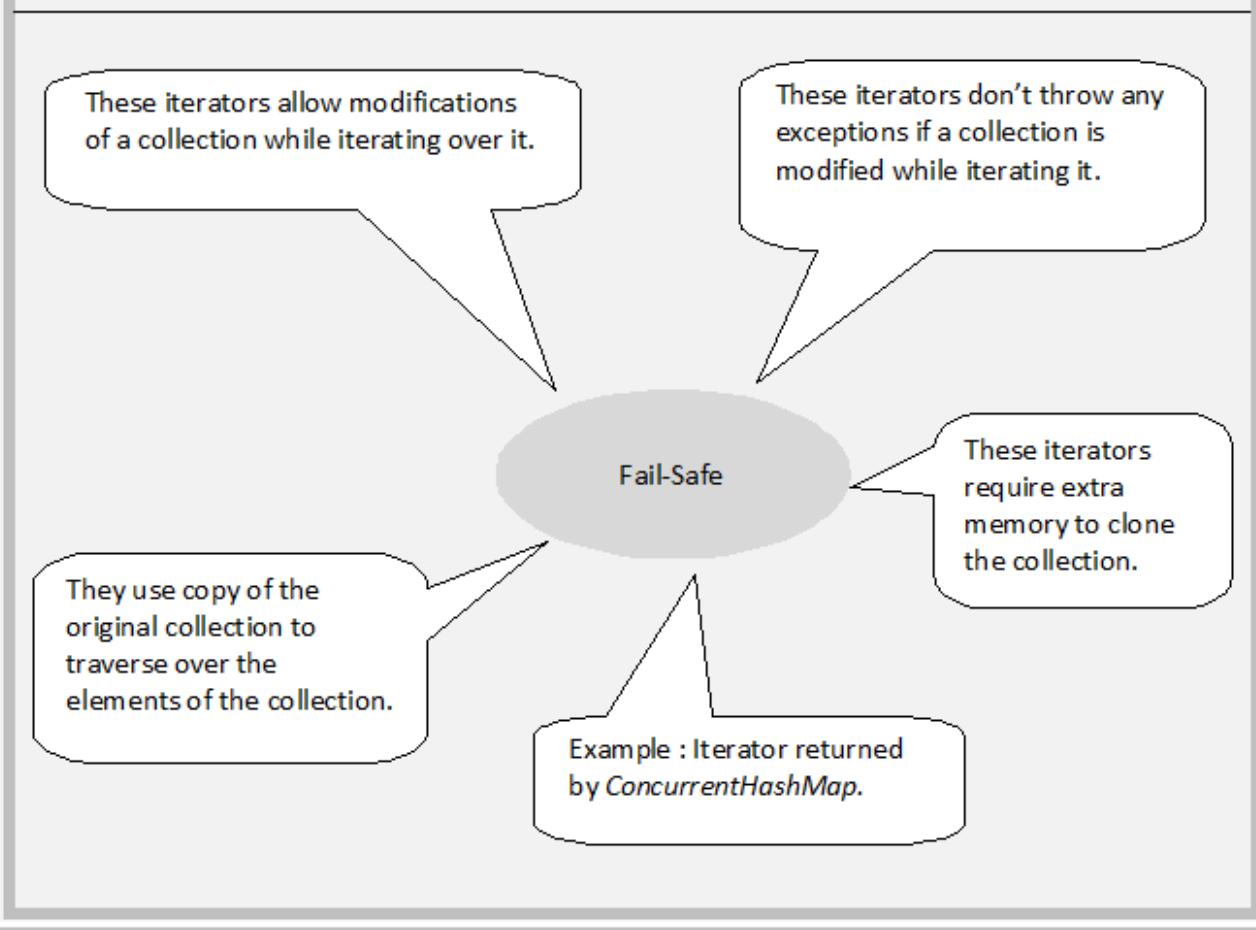
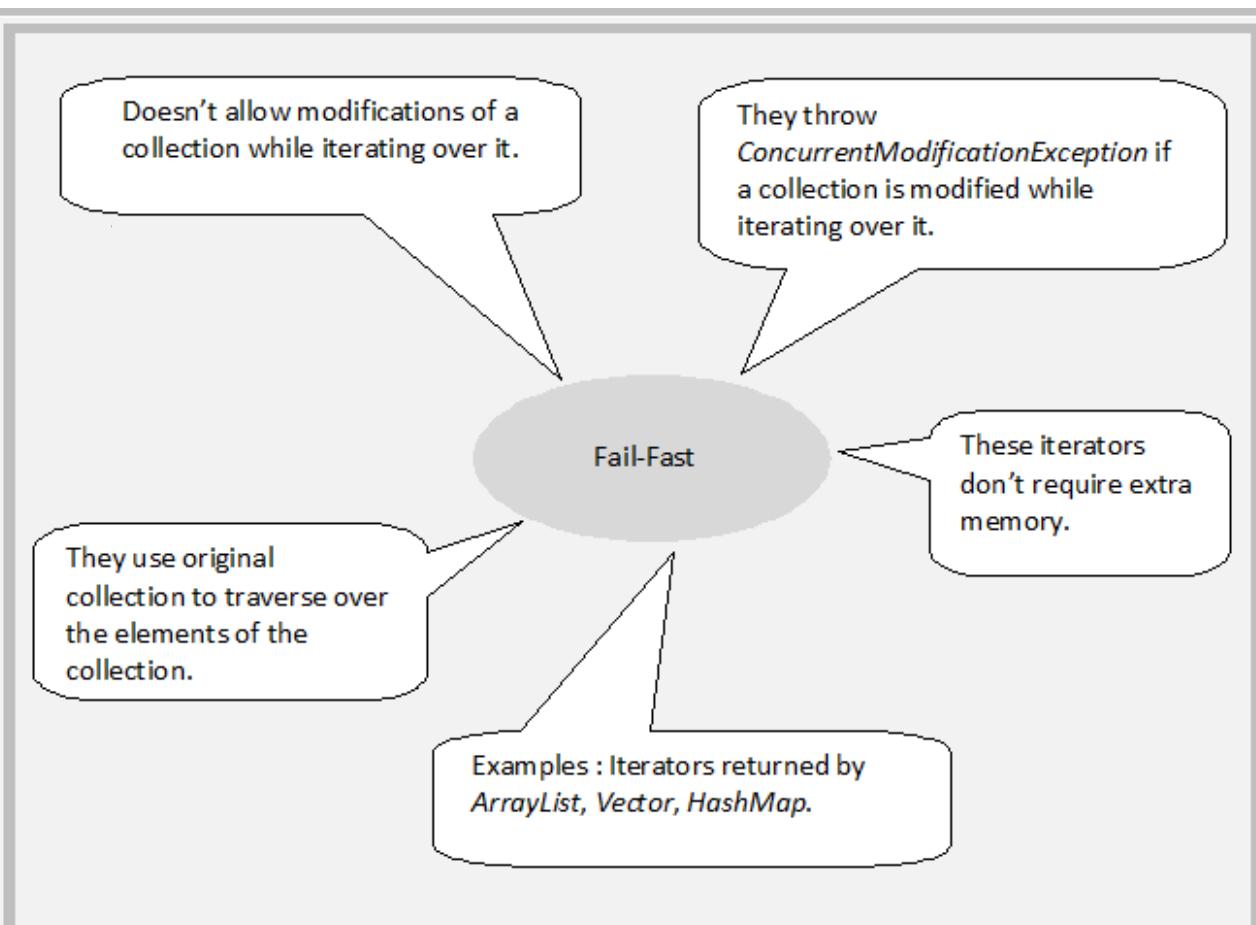
by *ArrayList*, *Vector*, *HashMap*.

These iterators don't throw any exceptions if a collection is modified while iterating over it.

They use copy of the original collection to traverse over the elements of the collection.

These iterators require extra memory to clone the collection.

Ex : Iterator returned by *ConcurrentHashMap*.



Multithreading

- **1. What is Thread in java?**
- Threads consumes CPU in best possible manner, hence enables multi processing. Multi threading reduces idle time of CPU which improves performance of application.
- Thread are light weight process.
- A thread class belongs to `java.lang` package.
- We can create multiple threads in java, even if we don't create any Thread, one Thread at least do exist i.e. main thread.
- Multiple threads run parallelly in java.
- Threads have their own stack.
- Advantage of Thread : Suppose one thread needs 10 minutes to get certain task, 10 threads used at a time could complete that task in 1 minute, because threads can run parallelly.

- **2. What is difference between Process and Thread in java?**
- Answer. One process can have multiple Threads,
- Thread are subdivision of Process. One or more Threads runs in the context of process. Threads can execute any part of process. And same part of process can be executed by multiple Threads.
- Processes have their own copy of the data segment of the parent process while Threads have direct access to the data segment of its process.
- Processes have their own address while Threads share the address space of the process that created it.
- Process creation needs whole lot of stuff to be done, we might need to copy whole parent process, but Thread can be easily created.
- Processes can easily communicate with child processes but interprocess communication is difficult. While, Threads can easily

communicate with other threads of the same process using [wait\(\)](#) and [notify\(\)](#) methods.

- In process all threads share system resource like heap Memory etc. while Thread has its own stack.
- Any change made to process does not affect child processes, but any change made to thread can affect the behavior of the other threads of the process.
- [Example to see where threads are created on different processes and same process.](#)

3. How to implement Threads in java?

Answer. This is very basic threading question. Threads can be created in two ways i.e. by [implementing java.lang.Runnable interface or extending java.lang.Thread class](#) and then extending run method.

Thread has its own variables and methods, it lives and dies on the heap. [But a thread of execution is an individual process that has its own call stack](#). Thread are lightweight process in java.

1. Thread creation by implementing `java.lang.Runnableinterface`.

We will create object of class which implements Runnable interface :

```
MyRunnable runnable=new MyRunnable();

Thread thread=new Thread(runnable);
```

2) And then create Thread object by calling constructor and passing reference of Runnable interface i.e. runnable object :

```
Thread thread=new Thread(runnable);
```

4. We should implement Runnable interface or extend Thread class. What are differences between implementing Runnable and extending Thread?

Answer. Well the answer is you must [extend Thread](#) only when you are looking to modify run() and other methods as well. If you are simply

looking to modify only the run() method [implementing Runnable](#) is the best option (Runnable interface has only one abstract method i.e. run()).

[Differences between implementing Runnable interface and extending Thread class -](#)

1. Multiple inheritance is not allowed in java : When we [implement Runnable](#) interface we can extend another class as well, but if we extend Thread class we cannot extend any other class because java does not allow multiple inheritance. So, same work is done by implementing Runnable and [extending Thread](#) but in case of implementing Runnable we are still left with option of extending some other class. So, it's better to implement Runnable.
2. [Thread safety](#) : When we implement Runnable interface, same object is shared amongst multiple threads, but when we extend Thread class each and every thread gets associated with new object.
3. Inheritance (Implementing Runnable is lightweight operation) : When we extend Thread unnecessary all Thread class features are inherited, but when we implement Runnable interface no extra feature are inherited, as Runnable only consists only of one abstract method i.e. run() method. So, implementing Runnable is lightweight operation.
4. Coding to interface : Even java recommends coding to interface. So, we must implement Runnable rather than extending thread. Also, Thread class implements Runnable interface.
5. Don't extend unless you wanna modify fundamental behaviour of class, Runnable interface has only one abstract method i.e. run() : We must [extend Thread](#) only when you are looking to modify run() and other methods as well. If you are simply looking to modify only the run() method [implementing Runnable](#) is the best option (Runnable interface has only one abstract method i.e. run()). We must not extend Thread class unless we're looking to modify fundamental behaviour of Thread class.
6. Flexibility in code when we implement Runnable : When we extend Thread first of all all thread features are inherited and our class becomes direct subclass of Thread , so whatever action we are doing is in Thread class. But, when we implement Runnable we create a new thread and pass runnable object as parameter,we could pass runnable object to executorService & much more. So, we have more options when we implement Runnable and our code becomes more flexible.
7. ExecutorService : If we implement Runnable, we can start multiple threads created on runnable object with ExecutorService (because we can start Runnable object with new threads), but not in the case when we extend Thread (because thread can be started only once).

5. How can you say Thread behaviour is unpredictable?
(Important)

Answer. The solution to question is quite simple, [Thread behaviour is unpredictable](#) because execution of Threads depends on Thread scheduler, thread scheduler may have different implementation on different platforms like windows, unix etc. Same threading program may produce different output in subsequent executions even on same platform.

To achieve we are going to create 2 threads on same Runnable Object, create for loop in run() method and start both threads. There is no surety that which threads will complete first, both threads will enter anonymously in for loop.

6. When threads are not lightweight process in java?

Answer. Threads are [lightweight process](#) only if threads of same process are executing concurrently. But if threads of different processes are executing concurrently then threads are [heavy weight process](#).

7. How can you ensure all threads that started from main must end in order in which they started and also main should end in last? (Important)

Answer. Interviewers tend to know interviewees knowledge about Thread methods. So this is time to prove your point by answering correctly. We can use [join\(\) method](#) to ensure all threads that started from main must end in order in which they started and also main should end in last. In other words waits for this thread to die. Calling join() method internally calls join(0);

We can use **join() method** to ensure all threads that started from main must end in order in which they started and also main should end in last. In other words **waits for this thread to die**.

Calling join() method internally calls join(0);

[**10 salient features of join\(\) method >**](#)

- **Definition : join()** We can use **join() method** to ensure all threads that started from main must end in order in which they started and also main should end in last.In other words **waits for thread to die on which thread has been called.**
- **Exception : join()** method **throws InterruptedException**, in our case we have thrown exception.
- **instance method : join()** is a **instance method**, hence we need to have **thread instance** for calling this method.
- **Thread state : when join() method is called on thread it goes from running to waiting state. And wait for thread to die.**
- **Not a native method :** implementation of **join()** method is provided in **java.lang.Thread class.**

Let's see definition of join() method as given in **java.lang.Thread** -

```
public final void join() throws InterruptedException;
```

- **synchronized block :** thread **need not to acquire object lock** before calling **join()** method i.e. join() method **can be called from outside synchronized block.**
 - **Waiting time : join() method have got few options.**
 1. **join()** : Waits for this thread to die.
- ```
public final void join() throws InterruptedException;
```
- This method internally calls **join(0)**. And timeout of 0 means to wait forever;
2. **join(long millis)** - Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

```
public static native void join(long millis) throws
InterruptedException;
```

3. **join(long millis, int nanos)** - Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

```
public static native void join(long millis,int nanos) throws
InterruptedException;
```

- **Belongs to which class : join() method belongs to java.lang.Thread class.**

### **8. Write a program to demonstrate the join()**

- To achieve we are going to create 2 threads on Runnable Object, create for loop in run() method and start both threads. After starting each Thread call join() method on them to ensure they end in order in which they has started.
- 

#### **• Full Program to show usage of join() method>**

```
class MyRunnable implements Runnable{
 public void run(){
 System.out.println("in run() method");
 for(int i=0;i<5;i++){
 System.out.println("i="+i+
,ThreadName="+Thread.currentThread().getName());
 }
 }
}
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String...args) throws
InterruptedException{
 System.out.println("In main() method");
 MyRunnable runnable=new MyRunnable();
 Thread thread1=new Thread(runnable);
 Thread thread2=new Thread(runnable);
```

```

 thread1.start();
thread1.join();

 thread2.start();
thread2.join();

 System.out.println("end main() method");
 }
}

/*OUTPUT
In main() method
in run() method
i=0 ,ThreadName=Thread-0
i=1 ,ThreadName=Thread-0
i=2 ,ThreadName=Thread-0
i=3 ,ThreadName=Thread-0
i=4 ,ThreadName=Thread-0
in run() method
i=0 ,ThreadName=Thread-1
i=1 ,ThreadName=Thread-1
i=2 ,ThreadName=Thread-1
i=3 ,ThreadName=Thread-1
i=4 ,ThreadName=Thread-1
end main() method
*/

```

- If we note output, all threads ended in order in which they were called and main thread has ended last.
- First, main thread was called, it started Thread1 and then we called join() method on Thread1, once Thread1 ended main thread started Thread2 and we called join() method on Thread2, once Thread2 ended main thread also ended.
- **In short - calling `thread1.join()` made main thread to wait until Thread-1 dies.**

## **9. What are the versions of join() method?**

**join()** : Waits for this thread to die.

```
public final void join() throws InterruptedException;
```

This method internally calls **join(0)**. And timeout of 0 means to wait forever;

**join(long millis)** - Waits at most millis milliseconds for this thread to die. A timeout of 0 means to wait forever.

```
public static native void join(long millis) throws
InterruptedException;
```

**join(long millis, int nanos)** - Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.

```
public static native void join(long millis,int nanos) throws
InterruptedException;
```

Let's create a program to use **join(long millis)** >

First, join(1000) will be called on Thread-1, **but once 1000 millisec are up, main thread can resume and start thread2 (main thread won't wait for Thread-1 to die).**

```
class MyRunnable implements Runnable{
 public void run(){
 System.out.println("in run() method");
 for(int i=0;i<5;i++){
 try {
 Thread.sleep(500);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println("i="+i+
,ThreadName="+Thread.currentThread().getName());
 }
 }
}
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String...args) throws
InterruptedException{
 System.out.println("In main() method");
 MyRunnable runnable=new MyRunnable();
 Thread thread1=new Thread(runnable);
 Thread thread2=new Thread(runnable);
 thread1.start();
 thread1.join(1000); //once 1000 millisec are up, main
thread can resume and start thread2.
 thread2.start();
 thread2.join();
```

```

 System.out.println("end main() method");
 }
}
/*OUTPUT
In main() method
in run() method
i=0 ,ThreadName=Thread-0
i=1 ,ThreadName=Thread-0
in run() method
i=2 ,ThreadName=Thread-0
i=0 ,ThreadName=Thread-1
i=1 ,ThreadName=Thread-1
i=3 ,ThreadName=Thread-0
i=2 ,ThreadName=Thread-1
i=4 ,ThreadName=Thread-0
i=3 ,ThreadName=Thread-1
i=4 ,ThreadName=Thread-1
end main() method
*/

```

#### **10. What is difference between starting thread with run() and start() method? (Important)**

Answer. This is quite interesting question, it might confuse you a bit and at time may make you think is there really any [difference between starting thread with run\(\) and start\(\) method.](#)

When you call start() method, main thread internally calls run() method to start newly created Thread, so run() method is ultimately called by newly created thread.

When you call run() method main thread rather than starting run() method with newly thread it start run() method by itself.

**Let's use start() method to start a thread>**

```

class MyRunnable implements Runnable{
 public void run(){ //overrides Runnable's run() method
 System.out.println("in run() method");
 System.out.println("currentThreadName= "+
Thread.currentThread().getName());
 }
}

```

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String args[]){
 System.out.println("currentThreadName= "+ Thread.currentThread().getName());
 MyRunnable runnable=new MyRunnable();
 Thread thread=new Thread(runnable);
 thread.start();
 }
}
/*OUTPUT
currentThreadName= main
in run() method
currentThreadName= Thread-0
*/

```

If we note output, when we called start() from main thread, **run() method was called by new Thread (i.e. Thread-0)**.

### Let's use run() method to start a thread>

```
class MyRunnable implements Runnable{
 public void run(){ //overrides Runnable's run() method
 System.out.println("in run() method");
 System.out.println("currentThreadName= "+ Thread.currentThread().getName());
 }
}
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String args[]){
 System.out.println("currentThreadName= "+ Thread.currentThread().getName());
 MyRunnable runnable=new MyRunnable();
 Thread thread=new Thread(runnable);
 thread.run();
 }
}
/*OUTPUT
currentThreadName= main
in run() method
currentThreadName= main
*/

```

If we note output, when we called run() from main thread, **run()** method was **called by main Thread**, not by newly created thread (i.e. **Thread-0**).

## **11. What is significance of using Volatile keyword?** **(Important)**

To Understand example of volatile keyword in java let's go back to [Singleton pattern in Java](#) and see [double checked locking in Singleton](#) with Volatile and without the volatile keyword in java.

```
/*
 * Java program to demonstrate where to use Volatile
 * keyword in Java.
 * In this example Singleton Instance is declared as
 * volatile variable to ensure
 * every thread see updated value for _instance.
 *
 * @author Javin Paul
 */
public class Singleton{
private static volatile Singleton _instance; //volatile
variable

public static Singleton getInstance() {

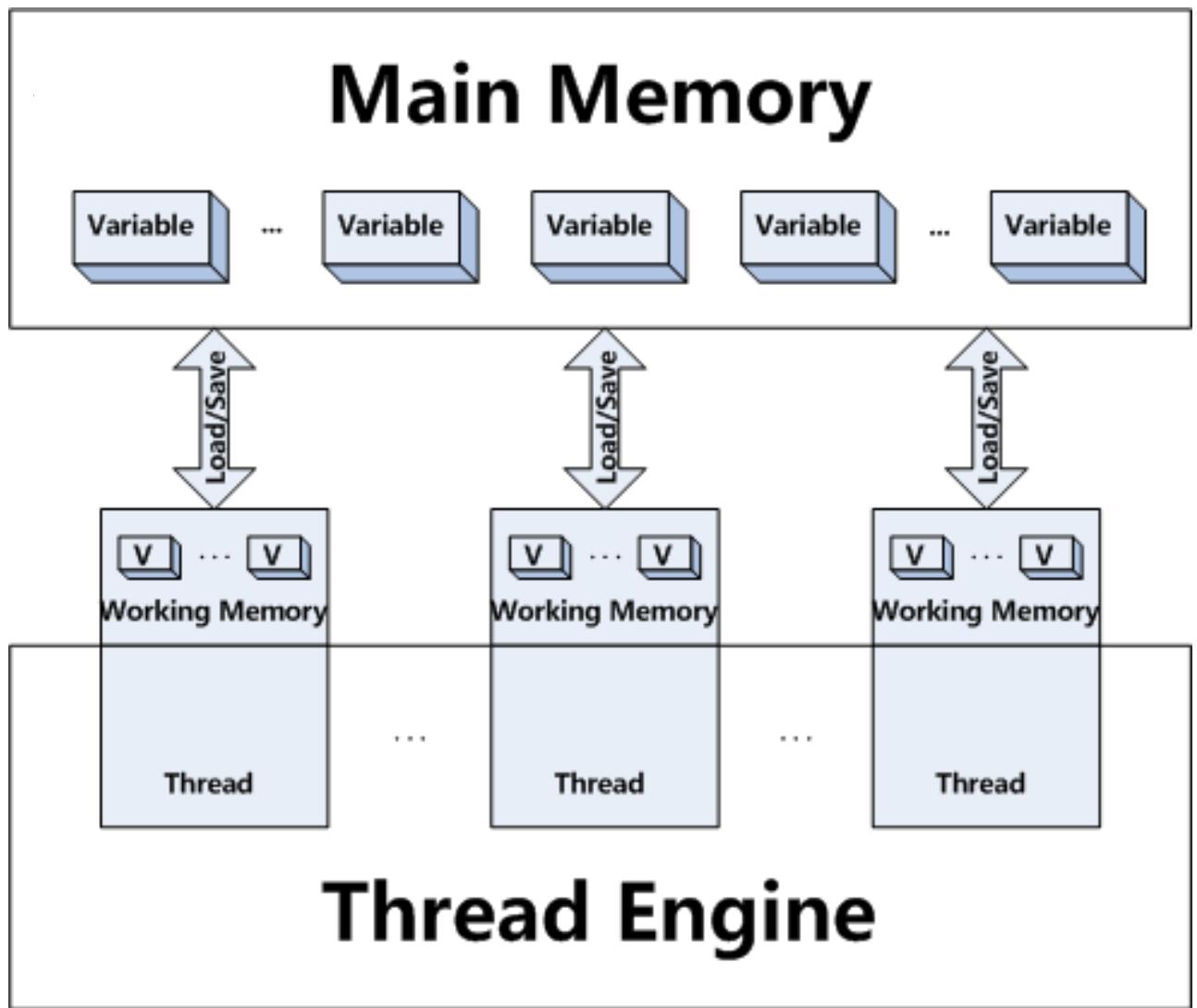
 if(_instance == null){
 synchronized(Singleton.class) {
 if(_instance == null)
 _instance = new Singleton();
 }
 }
 return _instance;
}
}
```

If you look at the code carefully you will be able to figure out:

- 1) We are only creating instance one time
- 2) We are creating instance lazily at the time of the first request comes.

If we do not make the `_instance` variable volatile than the Thread which is creating instance of Singleton is not able to communicate other thread, that instance has been created until it comes out of the Singleton block, so if Thread A is creating Singleton instance and just after creation lost the CPU, all

other thread will not be able to see value of `_instance` as not null and they will believe its still null.



Why? because reader threads are not doing any locking and until writer thread comes out of synchronized block, memory will not be synchronized and value of `_instance` will not be updated in main memory. With Volatile keyword in Java, this is handled by Java itself and such updates will be visible by all reader threads.

So in Summary apart from synchronized keyword in Java, volatile keyword is also used to communicate the content of memory between threads.

Let's see another example of volatile keyword in Java

- most of the time while writing game we use a variable `bExit` to check whether user has pressed exit button or not, value of this variable is updated in [event thread](#) and checked in game thread, So if we don't use volatile keyword with this variable, Game Thread might miss update from event handler thread if it's not synchronized in Java already. volatile keyword in java guarantees that value of the volatile variable will always be read from main memory and "*happens-before*" relationship in Java Memory model will ensure that content of memory will be communicated to different threads.

```
private boolean bExit;

while(!bExit) {
 checkUserPosition();
 updateUserPosition();
}
```

In this code example, One Thread (Game Thread) can cache the value of "bExit" instead of getting it from [main memory](#) every time and if in between any other thread (Event handler Thread) changes the value; it would not be visible to this thread. Making boolean variable "bExit" as volatile in java ensures this will not happen.

Also, If you have not read already then I also suggest you read the topic about volatile variable from [Java Concurrency in Practice](#) book by Brian Goetz, one of the must read to truly understand this complex concept.

## When to use Volatile variable in Java

- One of the most important thing in learning of volatile keyword is understanding when to use volatile variable in Java. Many [programmer](#) knows what is volatile variable and how does it work but they never really used volatile for any practical purpose. Here are couple of example to demonstrate when to use Volatile keyword in Java:
- 1) You can use Volatile variable if you want to read and write long and [double](#) variable atomically. long and double both are [64 bit](#) data type and by default writing of long and double is not atomic and platform dependence. Many platform perform write in long and double variable 2 step, writing 32 bit

in each step, due to this it's possible for a Thread to see 32 bit from two different write. You can avoid this issue by making long and double variable volatile in Java.

2) A volatile variable can be used as an alternative way of achieving [synchronization in Java](#) in some cases, like Visibility. with volatile variable, it's guaranteed that all reader thread will see updated value of the volatile variable once write operation completed, without volatile keyword different reader thread may see different values.

3) volatile variable can be used to inform the compiler that a particular field is subject to be accessed by multiple threads, which will prevent the compiler from doing any reordering or any kind of optimization which is not desirable in a multi-threaded environment. Without volatile variable compiler can re-order the code, free to cache value of volatile variable instead of always reading from main memory. like following example without volatile variable may result in an [infinite loop](#)

```
private boolean isActive = thread;
public void printMessage() {
 while(isActive){
 System.out.println("Thread is Active");
 }
}
```

without the *volatile modifier*, it's not guaranteed that one [Thread](#) sees the updated value of `isActive` from other thread. The compiler is also free to cache value of `isActive` instead of reading it from main memory in every iteration. By making `isActive` a volatile variable you avoid these issue.

4) Another place where a volatile variable can be used is to fixing double checked locking in Singleton pattern. As we discussed in [Why should you use Enum as Singleton](#) that double checked locking was broken in Java 1.4 environment.

Java allows threads to access shared variables. As a rule, to ensure that shared variables are consistently updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock that enforces mutual exclusion for those shared variables.

If a field is declared [volatile](#), in that case the Java memory model ensures that all threads see a consistent value for the variable.

Few small questions>

**9. Can we have volatile methods in java?**

1. No, volatile is only a keyword, can be used only with variables.

**10. Can we have synchronized variable in java?**

1. No, synchronized can be used only with methods i.e. in method declaration or synchronized blocks in java.

**11. Can you again start Thread?**

No, [we cannot start Thread again](#), doing so will throw `runtimeException java.lang.IllegalThreadStateException`. The reason is once `run()` method is executed by Thread, it goes into [dead state](#).

Let's take an example-

Thinking of starting thread again and calling `start()` method on it (which internally is going to call `run()` method) for us is some what like asking dead man to wake up and run. As, after completing his life person goes to dead state.

**12. What is race condition in multithreading and how can we solve it? (Important)**

Answer. This is very important question, this forms the core of multi threading, you should be able to explain about [race condition in detail](#). When more than one thread try to access same resource without synchronization causes race condition.

So we can [solve race condition](#) by using either [synchronized block or synchronized method](#). When no two threads can access same resource at a time phenomenon is also called as mutual exclusion.

Few sub questions>

What if two threads try to read same resource without [synchronization](#)?

When two threads try to read on same resource without synchronization, it's never going to create any problem.

What if two threads try to write to same resource without [synchronization](#)?

When two threads try to write to same resource without synchronization, it's going to create synchronization problems.

**13. What is deadlock in multithreading? Write a program to form DeadLock in multi threading and also how to solve DeadLock situation. What measures you should take to avoid deadlock? (Important)**

Deadlock is a situation where two threads are waiting for each other to release lock held by them on resources.

But how [deadlock](#) could be formed :

Thread-1 acquires lock on String.class and then calls [sleep\(\)](#) method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and Thread-2 acquires lock on Object.class then calls sleep() method and now it waits for Thread-1 to release lock on String.class.

Conclusion:

Now, Thread-1 is waiting for Thread-2 to release lock on Object.class and Thread-2 is waiting for Thread-1 to release lock on String.class and deadlock is formed.

```
//Code called by Thread-1

public void run() {

 synchronized (String.class) {

 Thread.sleep(100);
 }
}
```

```
synchronized (Object.class) {
}
}
}
}
```

//Code called by Thread-2

```
public void run() {

 synchronized (Object.class) {

 Thread.sleep(100);

 synchronized (String.class) {

 }
 }
}
```

Few important measures to avoid [Deadlock](#) >

1. Lock specific member variables of class rather than locking whole class: We must try to lock specific member variables of class rather than locking whole class.
1. Use join() method: If possible try to use join() method, although it may restrain us from taking full advantage of multithreading

environment because threads will start and end sequentially, but it can be handy in avoiding deadlocks.

1. If possible try avoid using nested synchronization blocks.

#### **14. Why wait(), notify() and notifyAll() are in Object class and not in Thread class? (Important)**

1. Every Object has a monitor, acquiring that monitors allow thread to hold lock on object. But Thread class does not have any monitors.
1. wait(), notify() and notifyAll() are called on objects only > When wait() method is called on object by thread it waits for another thread on that object to release object monitor by calling notify() or notifyAll() method on that object.

When notify() method is called on object by thread it notifies all the threads which are waiting for that object monitor that object monitor is available now. So, this shows that wait(), notify() and notifyAll() are called on objects only.

Now, Straight forward question that comes to mind is how thread acquires object lock by

acquiring object monitor? Let's try to understand this basic concept in detail?

1. Wait(), notify() and notifyAll() method being in Object class allows all the threads created on that object to communicate with other. .
2. As multiple threads exists on same object. Only one thread can hold object monitor at a time. As a result thread can notify other threads of same object that lock is available now. But, thread having these methods does not make any sense because multiple threads exists on object its not other way around (i.e. multiple objects exists on thread).
3. Now let's discuss one hypothetical scenario, what will happen if Thread class contains wait(), notify() and notifyAll() methods?

Having wait(), notify() and notifyAll() methods means Thread class also must have their monitor.

Every thread having their monitor will create few problems -

> Thread communication problem.

- >Synchronization on object won't be possible- Because object has monitor, one object can have multiple threads and thread hold lock on object by holding object monitor. But if each thread will have monitor, we won't have any way of achieving synchronization.
- >>Inconsistency in state of object (because synchronization won't be possible).

**15. Is it important to acquire object lock before calling wait(), notify() and notifyAll()?**

Answer.Yes, it's mandatory to acquire object lock before calling these methods on object. As discussed above wait(), notify() and notifyAll() methods are always called from **Synchronized block** only, and as soon as thread enters synchronized block it acquires object lock (by holding object monitor). If we call these methods without acquiring object lock i.e. from outside synchronize block then java.lang.IllegalMonitorStateException is thrown at runtime.

Wait() method needs to be enclosed in try-catch block, because it throws compile time exception i.e. InterruptedException.

**16. Have you ever generated thread dumps or analyzed Thread Dumps? (Important)**

[\*\*VisualVM\*\*](#) is most popular way to generate Thread Dump and is most widely used by developers. It's important to understand usage of VisualVM for in depth knowledge of VisualVM. I'll recommend every developer must understand this topic to become master in multi threading.

It helps us in analyzing threads performance, [\*\*thread states\*\*](#), CPU consumed by threads, garbage collection and much more.

Link:

[\*\*VisualVM link for understanding\*\*](#)

[\*\*jstack\*\*](#) is very easy way to generate Thread dump and is widely used by developers. I'll recommend every developer must understand this topic to become master in multi threading. For creating Thread dumps we need not to download any jar or any extra software.

[\*\*jStack link for understanding\*\*](#)

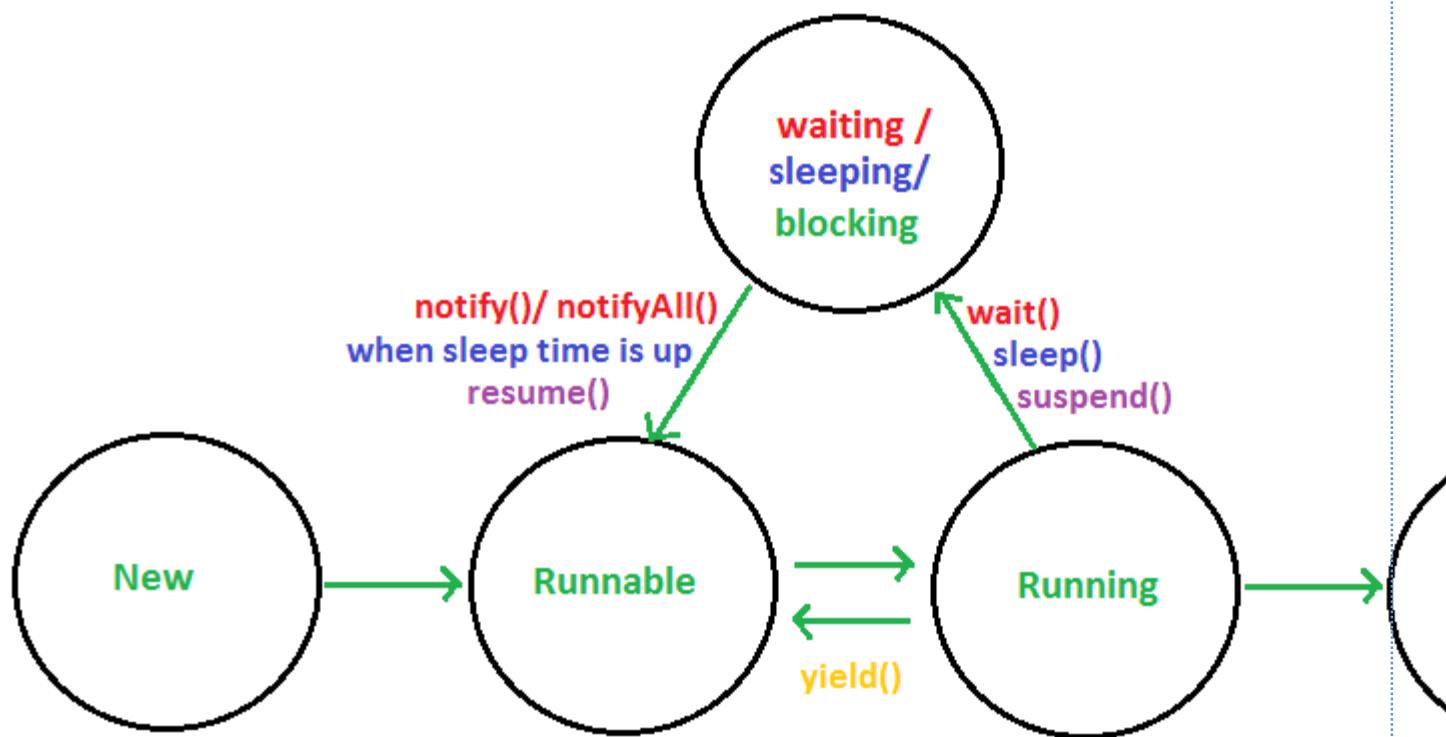
**17. What is life cycle of Thread, explain thread states?**  
**(Important)**

Thread states/ Thread life cycle is very basic question, before going deep into concepts we must understand Thread life cycle.

Thread have following states >

- New
- Runnable
- Running
- Waiting/blocked/sleeping
- Terminated (Dead)

Thread states/ Thread life cycle in diagram >



**Fig. THREAD STATES**

Thread states in detail >

New : When instance of thread is created using new operator it is in new state, but the start() method has not been invoked on the thread yet, thread is not eligible to run yet.

Runnable : When start() method is called on thread it enters runnable state.

Running : Thread scheduler selects thread to go from runnable to running state. In running state Thread starts executing by entering run() method.

Waiting/blocked/sleeping : In this state a thread is not eligible to run.

> Thread is still alive, but currently it's not eligible to run. In other words.

> How can Thread go from running to waiting state?

By calling wait()[method](#) thread go from running to waiting state. In waiting state it will wait for other threads to release object monitor/lock.

> How can Thread go from running to sleeping state?

By calling sleep()[method](#) thread go from running to sleeping state. In sleeping state it will wait for sleep time to get over.

Terminated (Dead) : A thread is considered dead when its run() method completes.

## **18. Are you aware of preemptive scheduling and time slicing?**

- In preemptive scheduling, the highest priority thread executes until it enters into the [waiting or dead state](#).
- In time slicing, a thread executes for a certain predefined time and then enters runnable pool. Than thread can enter running state when selected by thread scheduler.

## **22. What are daemon threads?**

[Daemon threads](#) are low priority threads which runs intermittently in background for doing garbage collection.

## 12 Few salient features of daemon() threads>

- Thread scheduler schedules these threads only when CPU is idle.
- Daemon threads are service oriented threads, they serve all other threads.
- These threads are created before user threads are created and die after all other user threads dies.
- Priority of daemon threads is always 1 (i.e. MIN\_PRIORITY).
- User created threads are non daemon threads.
- JVM can exit when only daemon threads exist in system.
- we can use isDaemon() method to check whether thread is daemon thread or not.
- we can use setDaemon(boolean on) method to make any user method a daemon thread.
- If setDaemon(boolean on) is called on thread after calling start() method than IllegalThreadStateException is thrown.
- You may like to see how daemon threads work, for that you can use VisualVM or jStack. I have provided Thread dumps over there which shows daemon threads which were intermittently running in background.

Some of the daemon threads which intermittently run in background are >

```
"RMI TCP Connection(3)-10.175.2.71" daemon"RMI TCP Connection(idle)" daemon"RMI Scheduler(o)" daemon"C2 CompilerThread1" daemon
```

```
"GC task thread#0 (ParallelGC)"
```

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
```

```
public class DaemonTest {
```

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

```

final Thread thread1=new Thread("Thread-1"){

 public void run() {

 System.out.println(Thread.currentThread().getName()+" has started");

 System.out.println(Thread.currentThread().getName()+" has ended");

 }

};

thread1.setDaemon(true); //setting thread to daemon.

System.out.println("is thread1 daemon thread : "

 +thread1.isDaemon()); //checking thread isDeamon ?

thread1.start(); //start daemon thread

}

}

/*
is thread1 daemon thread : true

Thread-1 has started

Thread-1 has ended

*/

```

### **23. Why suspend() and resume() methods are deprecated?**

Answer. Suspend() method is deadlock prone. If the target thread holds a lock on object when it is suspended, no thread can lock this object until the target thread is resumed. If the thread that would resume the target

thread attempts to lock this monitor prior to calling resume, it results in deadlock formation.

These deadlocksare generally called Frozen processes.

Suspend() method puts thread from running to waiting state. And thread can go from waiting to runnable state only when resume() method is called on thread. It is deprecated method.

Resume() method is only used with suspend() method that's why it's also deprecated method.

#### **24. Why destroy() methods is deprecated?**

Answer. This question is again going to check your in depth knowledge of thread methods i.e. destroy() method is deadlock prone. If the target thread holds a lock on object when it is destroyed, no thread can lock this object (Deadlock formed are similar to deadlock formed when suspend() and resume() methods are used improperly). It results in deadlock formation. These deadlocksare generally called Frozen processes.

Additionally you must know calling destroy() method on Threads throw runtimeException i.e. NoSuchMethodError. Destroy() method puts thread from running to dead state.

#### **25. As stop() method is deprecated, How can we terminate or stop infinitely running thread in java? (Important)**

Answer. This is very interesting question where interviewees thread basics will be tested. Interviewers tend to know user's knowledge about main thread's and thread invoked by main thread.

We will try to address the problem by creating new thread which will run infinitely until certain condition is satisfied and will be called by main Thread.

1. Infinitely running thread can be stopped using boolean variable.
2. Infinitely running thread can be stopped using interrupt() method.

Let's understand Why stop() method is deprecated :

Stopping a thread with Thread.stop() causes it to **release all of the monitors that it has locked**. If any of the objects previously protected by these monitors were in an

inconsistent state, the damaged objects become visible to other threads, which might lead to unpredictable behavior.

## **26. what is significance of yield() method, what state does it put thread in?**

[yield\(\)](#) is a native method it's implementation in java 6 has been changed as compared to its implementation java 5. As method is native it's implementation is provided by JVM.

In java 5, yield() method internally used to call [sleep\(\)](#) method giving all the other threads of same or higher priority to execute before yielded thread by leaving allocated CPU for time gap of 15 millisec.

But java 6, calling yield() method gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint. So, sometimes even after using yield() method, you may not notice any difference in output.

salient features of [yield\(\)](#) method >

- Definition : [yield\(\)](#) method when called on thread gives a hint to the thread scheduler that the current thread is willing to yield its current use of a processor. The thread scheduler is free to ignore this hint.
- [Thread state](#) : when yield() method is called on thread it goes from running to runnable state, not in waiting state. Thread is eligible to run but not running and could be picked by scheduler at anytime.
- Waiting time : yield() method stops thread for unpredictable time.
- Static method : yield() is a static method, hence calling Thread.yield() causes currently executing thread to yield.
- Native method : implementation of yield() method is provided by JVM.

Let's see definition of yield() method as given in java.lang.Thread -

```
public static native void yield();
```

- synchronized block : thread need not to acquire object lock before calling yield() method i.e. yield() method can be called from outside synchronized block.

**27. What is significance of sleep() method in detail, what state does it put thread in ?**

sleep() is a native method, its implementation is provided by JVM.

10 salient features of sleep() method >

- Definition : sleep() methods causes current thread to sleep for specified number of milliseconds (i.e. time passed in sleep method as parameter). Ex- Thread.sleep(10) causes currently executing thread to sleep for 10 millisec.
- Thread state : when sleep() is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.
- Exception : sleep() method must catch or throw compile time exception i.e. InterruptedException.
- Waiting time : sleep() method have got few options.
  1. sleep(long millis) - Causes the currently executing thread to sleep for the specified number of milliseconds

```
public static native void sleep(long millis) throws InterruptedException;
```

1. sleep(long millis, int nanos) - Causes the currently executing thread to sleep for the specified number of milliseconds plus the specified number of nanoseconds.

```
public static native void sleep(long millis,int nanos) throws InterruptedException;
```

- static method : sleep() is a static method, causes the currently executing thread to sleep for the specified number of milliseconds.

- Belongs to which class :[sleep\(\)](#) method belongs to `java.lang.Thread` class.
- synchronized block : thread need not to acquire object lock before calling sleep() method i.e. sleep() method can be called from outside synchronized block.

## 28. Difference between wait() and sleep() ? (Important)

Answer.

- Should be called from **synchronized block** :`wait()` method is always called from synchronized block i.e. [wait\(\)](#) method needs to lock object monitor before object on which it is called. But **sleep()** method can be called from outside synchronized block i.e. `sleep()` method doesn't need any object monitor.
- **IllegalMonitorStateException** : if `wait()` method is called without acquiring object lock than `IllegalMonitorStateException` is thrown at runtime, but `sleep()` method never throws such exception.
- Belongs to which **class** : [wait\(\)](#) method belongs to `java.lang.Object` class but `sleep()` method belongs to `java.lang.Thread` class.
- Called on object or **thread** : `wait()` method is called on objects but `sleep()` method is called on Threads not objects.
- **Thread state** : when `wait()` method is called on object, thread that held object's monitor goes from running to waiting state and can return to **runnable state only when `notify()` or `notifyAll()`** method is called on that object. And later thread scheduler schedules that thread to go from runnable to running state.

when `sleep()` is called on thread it goes from running to waiting state and can return to runnable state when sleep time is up.

- When called from **synchronized block** : when `wait()` method is called thread leaves the object lock. But `sleep()` method when called from synchronized block or method thread doesn't leave object lock.

**29. Does thread leaves object lock when wait() method is called?**

- Answer. When wait() method is called Thread leaves the object lock and goes from running to waiting state. Thread waits for other threads on same object to call notify() or notifyAll() and once any of notify() or notifyAll() is called it goes from waiting to runnable state and again acquires object lock.

**30. What will happen if we don't override run method?**

5. When we call start() method on thread, it internally calls run() method with newly created thread. So, if we don't override run() method newly created thread won't be called and nothing will happen.

```
class MyThread extends Thread {
 //don't override run() method
}

public class DontOverrideRun {
 public static void main(String[] args) {
 System.out.println("main has started.");
 MyThread thread1 = new MyThread();
 thread1.start();
 System.out.println("main has ended.");
 }
}
/*OUTPUT
main has started.
main has ended.
```

\*/

- As we saw in output, we didn't override run() method that's why on calling start() method nothing happened.

### **31. What will happen if we override start method?**

Answer. When we call start() method on thread, it internally calls run() method with newly created thread. So, if we override start() method, run() method will not be called until we write code for calling run() method.

```
class MyThread extends Thread {
 @Override
 public void run() {
 System.out.println("in run() method");
 }

 @Override
 public void start() {
 System.out.println("In start() method");
 }
}
public class OverrideStartMethod {
 public static void main(String[] args) {
 System.out.println("main has started.");

 MyThread thread1 = new MyThread();
 thread1.start();

 System.out.println("main has ended.");
 }
}
/*OUTPUT
main has started.
In start() method
main has ended.
```

### **32. Can we acquire lock on class? What are ways in which you can acquire lock on class?**

Answer. Yes, we can acquire lock on class's class object in 2 ways to acquire lock on class. Thread can acquire lock on class's class object by-

- Entering synchronized block** or Let's say there is one class MyClass. Now we can create synchronization block, and

parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized MyClass:

```
synchronized (MyClass.class) {
 //thread has acquired lock on MyClass's class object.
}
```

## **2. by entering static synchronized methods.**

```
public static synchronized void method1() {
 //thread has acquired lock on MyRunnable's class object.
}
```

As soon as thread entered Synchronization method, thread acquired lock on class's class object. Thread will leave lock when it exits static synchronized method.

## **33. Difference between object lock and class lock?**

| <u>Object lock</u>                                                                                                                                                                                                | <u>Class lock</u>                                                                                                                                                                                                        |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>Thread can acquire <u>object lock</u> by-</p> <ol style="list-style-type: none"> <li>1. Entering <b>synchronized block</b> or</li> <li>2. by entering <b>synchronized methods</b>.</li> </ol>                  | <p>Thread can acquire lock on <u>class's class object</u> by-</p> <ol style="list-style-type: none"> <li>1. Entering <b>synchronized block</b> or</li> <li>2. by entering <b>static synchronized methods</b>.</li> </ol> |
| <p><u>Multiple threads may exist on same object but only one thread of that object can enter synchronized method at a time.</u></p> <p><u>Threads on different object can enter same method at same time.</u></p> | <p>Multiple threads may exist on <u>same</u> or <u>different objects</u> of class but only one thread can enter <b>static synchronized method</b> at a time.</p>                                                         |

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Multiple objects of class may exist and every object has it's own lock.</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       | <b>Multiple objects of class may exist but there is always one class's class object lock available.</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| <p>First let's acquire <a href="#">object lock</a> by entering <b>synchronized block</b>.</p> <p>Example- Let's say there is one class <u>MyClass</u> and we have created it's object and reference to that object is <u>myClass</u>. Now we can create synchronization block, and parameter passed with synchronization tells which object has to be synchronized. In below code, we have synchronized object reference by <u>myClass</u>.</p> <pre><u>MyClass</u> myClass=<u>new</u> <u>Myclass</u>();     <b>synchronized</b> (<u>myClass</u>) { }</pre> <p>As soon thread entered Synchronization block, thread acquired object lock on object referenced by <u>myClass</u> (by acquiring object's monitor.)</p> <p>Thread will leave lock when it exits synchronized block.</p> | <p>First let's acquire lock on <a href="#">class's class object</a> by entering <b>synchronized block</b>.</p> <p>Example- Let's say there is one class <u>MyClass</u>. Now we can create synchronization block, and parameter passed with synchronization tells which class has to be synchronized. In below code, we have synchronized <u>MyClass</u></p> <pre><b>synchronized</b> (<u>MyClass.class</u>) {</pre> <p>As soon as thread entered Synchronization block, thread acquired <u>MyClass's class object</u>. Thread will leave lock when it exits synchronized block.</p> |
| <pre><b>public synchronized void</b> method1() { }</pre> <p>As soon as thread entered <b>Synchronization method</b>, thread acquired <a href="#">object lock</a>.</p> <p>Thread will leave lock when it exits synchronized method.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               | <pre><b>public static synchronized</b> <b>void</b> method1() {}</pre> <p>As soon as thread entered <b>static Synchronization method</b>, thread acquired lock on <a href="#">class's class object</a>.</p> <p>Thread will leave lock when it exits synchronized method.</p>                                                                                                                                                                                                                                                                                                         |

- 34.** Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time?

**Answer.** No, here when Thread-1 is in **synchronized method1()** it must be **holding lock on object's monitor** and will release lock on object's monitor only when it exits **synchronized method1()**. So, Thread-2 will have to **wait** for Thread-1 to release lock on object's monitor so that it could enter **synchronized method2()**.

Likewise, Thread-2 even cannot enter **synchronized method1()** which is being executed by Thread-1. Thread-2 will have to **wait** for Thread-1 to release lock on object's monitor so that it could enter **synchronized method1()**.

&gt;

```

class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
 method1();
 else
 method2();
 }
 synchronized void method1(){
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method1() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method1() ended");
 }

 synchronized void method2(){
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
}

```

```

 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method2() ended");
 }

}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String args[]) throws
InterruptedException{

 MyRunnable1 myRunnable1=new MyRunnable1();

 Thread thread1=new Thread(myRunnable1, "Thread-1");
 Thread thread2=new Thread(myRunnable1, "Thread-2");
 thread1.start();
 Thread.sleep(10); //Just to ensure Thread-1 starts before
Thread-2
 thread2.start();

 }
}
/*OUTPUT
Thread-1 in synchronized void method1() started
Thread-1 in synchronized void method1() ended
Thread-2 in synchronized void method2() started
Thread-2 in synchronized void method2() ended
*/

```

If you note output, when Thread-1 was in **synchronized method1()** it was **holding lock on object's monitor**. So, Thread-2 waited for Thread-1 to release lock on object's monitor to enter **synchronized method2()**.

- 35.** Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time?

**Answer.** No, here when Thread-1 is in **static synchronized method1()** it must be **holding lock on class class's object** and will release lock on class's class object only when it exits **static synchronized method1()**. So, Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method2()**.

Likewise, Thread-2 even cannot enter **static synchronized method1()** which is being executed by Thread-1. Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method1()**.

#### Program >

```

class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
 method1();
 else
 method2();
 }
 static synchronized void method1(){
 System.out.println(Thread.currentThread().getName()+
 " in static synchronized void method1()")
 started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()+
 " in static synchronized void method1()")
 ended");
 }

 static synchronized void method2(){
 System.out.println(Thread.currentThread().getName()+
 " in static synchronized void method2()")
 started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()+
 " in static synchronized void method2()")
 ended");
 }

}
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {

```

```

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

 MyRunnable1 myRunnable1=new MyRunnable1();

 Thread thread1=new Thread(myRunnable1, "Thread-1");
 Thread thread2=new Thread(myRunnable1, "Thread-2");
 thread1.start();
 Thread.sleep(10); //Just to ensure Thread-1 starts before
Thread-2
 thread2.start();

}

}
/*OUTPUT
Thread-1 in static synchronized void method1() started
Thread-1 in static synchronized void method1() ended
Thread-2 in static synchronized void method2() started
Thread-2 in static synchronized void method2() ended
*/

```

If you note output, when Thread-1 was in **static synchronized method1()** it was **holding lock on class class's object**. So, Thread-2 waited for Thread-1 to release lock on class's class object to enter **static synchronized method2()**.

**36.** Suppose you have 2 threads (Thread-1 and Thread-2) on same object. Thread-1 is in synchronized method1(), can Thread-2 enter static synchronized method2() at same time?

**Answer.** Yes, here when Thread-1 is in **synchronized method1()** it must be **holding lock on object's monitor** and Thread-2 can enter **static synchronized method2()** by acquiring lock on **class's class object**.

Program >

```

class MyRunnable1 implements Runnable{

 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
 method1();
 }
}

```

```

 else

 method2();

 }

 static synchronized void method1(){

 System.out.println(Thread.currentThread().getName()+
 " in static synchronized void method1()"

started");

 try {

 Thread.sleep(2000);

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 System.out.println(Thread.currentThread().getName()+
 " in static synchronized void method1()"

ended");

 }

}

synchronized void method2(){

 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method2() started");

 try {

 Thread.sleep(2000);

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method2() ended");

}

```

```

}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

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

 MyRunnable1 myRunnable1=new MyRunnable1();

 Thread thread1=new Thread(myRunnable1,"Thread-1");
 Thread thread2=new Thread(myRunnable1,"Thread-2");
 thread1.start();
 Thread.sleep(10);//Just to ensure Thread-1 starts before
Thread-2
 thread2.start();

}

}

/*OUTPUT
Thread-1 in static synchronized void method1() started
Thread-2 in synchronized void method2() started
Thread-1 in static synchronized void method1() ended
Thread-2 in synchronized void method2() ended
*/

```

If you note output, when Thread-1 was in **synchronized method1()** it was **holding lock on object's monitor** and Thread-2 entered **static synchronized method2()** by acquiring lock on [class's class object](#).

**37.** Suppose you have thread and it is in synchronized method and now can thread enter other synchronized method from that method?

**Answer.** Yes, here when thread is in **synchronized method** it must be **holding lock on object's monitor** and **using that lock** thread can **enter other synchronized method**

Program >

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 method1();
 }
 synchronized void method1(){
 System.out.println("synchronized method1() started");
 method2();
 System.out.println("synchronized method1() ended");
 }
 synchronized void method2(){
 System.out.println("in synchronized method2()");
 }
}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String args[]) throws
InterruptedException{

 MyRunnable1 myRunnable1=new MyRunnable1();
 Thread thread1=new Thread(myRunnable1,"Thread-1");
 thread1.start();

 }
}
/*OUTPUT
synchronized method1() started
in synchronized method2()
synchronized method1() ended
*/
```

If you note output, when thread was in **synchronized method1()** it was **holding lock on object's monitor** and **using that lock** thread **entered synchronized method2()**.

- 38.** Suppose you have thread and it is in static synchronized method and now can thread enter other static synchronized method from that method?

**Answer.** Yes, here when thread is in **static synchronized method** it must be **holding lock on class's class object** and **using that lock** thread can **enter other static synchronized method**.

Program >

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 method1();
 }

 static synchronized void method1(){
 System.out.println("static synchronized void method1() started");
 method2();
 System.out.println("static synchronized void method1() ended");
 }

 static synchronized void method2(){
 System.out.println("in static synchronized method2()");
 }

}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
```

```

public static void main(String args[]) throws
InterruptedException{
 MyRunnable1 myRunnable1=new MyRunnable1();
 Thread thread1=new Thread(myRunnable1, "Thread-1");
 thread1.start();
}
/*OUTPUT
static synchronized void method1() started
in static synchronized method2()
static synchronized void method1() ended
*/

```

If you note output, when thread was in **static synchronized method1()** it was **holding lock on class's class object** and **using that lock** thread entered **static synchronized method2()**.

**39.** Suppose you have thread and it is in static synchronized method and now can thread enter other non static synchronized method from that method?

**Answer.** Yes, here when thread is in **static synchronized method** it must be **holding lock on class's class object** and when it **enters synchronized method** it will **hold lock on object's monitor as well**.

So, now thread holds 2 locks (it's also called nested synchronization)-

>first one on **class's class object**.

>second one on **object's monitor** (This lock will be released when thread exits non static method)

Program >

```

class MyRunnable1 implements Runnable{
 @Override

```

```
public void run(){
 method1();
}

static synchronized void method1(){
 System.out.println("static synchronized method1() started");
 MyRunnable1 myRunnable1=new MyRunnable1();
 myRunnable1.method2();
 System.out.println("static synchronized method1() ended");
}

synchronized void method2(){
 System.out.println("in synchronized method2()");
}

}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

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

 MyRunnable1 myRunnable1=new MyRunnable1();
 Thread thread1=new Thread(myRunnable1, "Thread-1");
 thread1.start();

}
}

/*OUTPUT
```

```
static synchronized method1() started
in synchronized method2()
static synchronized method1() ended
*/
```

If you note output, when thread was in **static synchronized method1()** it was **holding lock on class's class object** and it entered **synchronized method2()** by acquiring **lock on object's monitor as well**.

40. Suppose you have thread and it is in synchronized method and now can thread enter other static synchronized method from that method?

**Answer.** Yes, here when thread is in synchronized method it must be holding **lock on object's monitor** and when it enters static synchronized method it will hold lock on **class's class object** as well.

So, now thread holds 2 locks (it's also called nested synchronization)-

>first one on **object's monitor**.

>second one on **class's class object**.(This lock will be released when thread exits static method).

Program >

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 method1();
 }

 synchronized void method1(){
 System.out.println("synchronized void method1() started");
 method2();
 System.out.println("synchronized void method1() ended");
 }
}
```

```
static synchronized void method2(){
 System.out.println("in static synchronized method2()");
}

}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

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

 MyRunnable1 myRunnable1=new MyRunnable1();
 Thread thread1=new Thread(myRunnable1, "Thread-1");
 thread1.start();

}

}

/*OUTPUT
synchronized void method1() started
in static synchronized method2()
synchronized void method1() ended
*/
```

If you note output, when thread was in synchronized method1() it was holding lock on object's monitor and when it entered static synchronized method2() it acquired lock on class's class object as well.

- 41.** Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in synchronized method1(), can Thread-2 enter synchronized method2() at same time?

**Answer.** Yes, here when Thread-1 is in **synchronized method1()** it must be **holding lock on object1's monitor**. Thread-2 will acquire lock on **object2's monitor** and enter **synchronized method2()**.

Likewise, Thread-2 even enter **synchronized method1()** as well which is being executed by Thread-1 (because threads are created on different objects).

Program >

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
 method1();
 else
 method2();
 }
 synchronized void method1(){
 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method1() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method1() ended");
 }

 synchronized void method2(){
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() ended");
 }
}
```

```

}

}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String args[]) throws
InterruptedException{
 MyRunnable1 object1=new MyRunnable1();
 MyRunnable1 object2=new MyRunnable1();

 Thread thread1=new Thread(object1,"Thread-1");
 Thread thread2=new Thread(object2,"Thread-2");
 thread1.start();
 Thread.sleep(10); //Just to ensure Thread-1 starts before
Thread-2
 thread2.start();

 }
}

/*OUTPUT
Thread-1 in synchronized void method1() started
Thread-2 in synchronized void method2() started
Thread-1 in synchronized void method1() ended
Thread-2 in synchronized void method2() ended
*/

```

If you note output, when Thread-1 was in **synchronized method1()** it was **holding lock on object1's monitor**. Thread-2 acquired lock on **object2's monitor** and entered **synchronized method2()**.

42. Suppose you have 2 threads (Thread-1 on object1 and Thread-2 on object2). Thread-1 is in static synchronized method1(), can Thread-2 enter static synchronized method2() at same time?

**Answer.** **No**, it might confuse you a bit that threads are created on different objects. But, not to forgot that **multiple objects may exist but there is always one class's class object lock available**.

Here, when Thread-1 is in **static synchronized method1()** it must be **holding lock on class class's object** and will release lock on class's class object only when it exits **static synchronized method1()**. So, Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method2()**.

## Program >

```

class MyRunnable1 implements Runnable{

 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
 method1();
 else
 method2();
 }

 static synchronized void method1(){
 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method1() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method1() ended");
 }

 static synchronized void method2(){
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 }
}

```

```

 e.printStackTrace();
 }

 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() ended");

}

}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

public class MyClass {

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

 MyRunnable1 object1=new MyRunnable1();

 MyRunnable1 object2=new MyRunnable1();

 Thread thread1=new Thread(object1,"Thread-1");
 Thread thread2=new Thread(object2,"Thread-2");
 thread1.start();
 Thread.sleep(10);//Just to ensure Thread-1 starts before
Thread-2

 thread2.start();

 }
}
/*OUTPUT
Thread-1 in synchronized void method1() started
Thread-1 in synchronized void method1() ended
Thread-2 in synchronized void method2() started

```

**Thread-2 in synchronized void method2() ended**

\*/

If you note output, when Thread-1 was in **static synchronized method1()** it was **holding lock on class's class object**. So, Thread-2 waited for Thread-1 to release lock on [class's class object](#) so that it could enter **static synchronized method2()**.

**Likewise**, Thread-2 even cannot enter **static synchronized method1()** which is being executed by Thread-1. Thread-2 will have to [wait](#) for Thread-1 to release lock on [class's class object](#) so that it could enter **static synchronized method1()**.

### **43. Difference between wait() and wait(long timeout), What are thread states when these method are called?**

**Answer.**

| <a href="#"><b>wait()</b></a>                                                                                                                                                                                                      | <a href="#"><b>wait(long timeout)</b></a>                                                                                                                                                                                                                                                    |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| When <a href="#">wait()</a> method is called on object, it causes causes the current thread to wait until another thread invokes the <a href="#">notify()</a> or <a href="#">notifyAll()</a> method for this object.               | <b>wait(long timeout)</b> - Causes the current thread to wait until either another thread invokes the <a href="#">notify()</a> or <a href="#">notifyAll()</a> methods for this object, or a specified timeout time has elapsed.                                                              |
| <b>When wait() is called</b> on object - Thread enters from <a href="#">running to waiting state</a> .<br><br>It <a href="#">waits</a> for some other thread to call <a href="#">notify</a> so that it could enter runnable state. | <b>When wait(1000) is called</b> on object - Thread enters from <a href="#">running to waiting state</a> . Then even if <a href="#">notify()</a> or <a href="#">notifyAll()</a> is not called after timeout time has elapsed thread will go from <a href="#">waiting to runnable state</a> . |

### **44. Can a constructor be synchronized?**

No, constructor cannot be synchronized. Because constructor is used for instantiating object, when we are in constructor object is under creation. So, until object is not instantiated it does not need any synchronization.

**Enclosing** constructor in synchronized block will generate compilation error.

Using synchronized in **constructor definition** will also show compilation error.

**COMPILATION ERROR = Illegal modifier for the constructor in type**

**ConstructorSynchronizeTest;** only public, protected & private are permitted

**Though we can use synchronized block inside constructor.**

*Using synchronized in constructor definition will also show compilation error >*

**COMPILATION ERROR = Illegal modifier for the constructor in type**

**ConstructorSynchronizeTest;** only public, protected & private are permitted

```

3 public class ConstructorSynchronizeTest {
4
5 /*using synchronized in constructor definition causes compilation error
6 * ERROR= Illegal modifier for the constructor in type ConstructorSynchronizeTest;
7 * only public, protected & private are permitted
8 */
9 public synchronized ConstructorSynchronizeTest() {
10
11 }
12 }
```

*Though we can use synchronized block inside constructor >*

```

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class ConstructorSynchronizeTest {

 //constructor
 public ConstructorSynchronizeTest() {
 synchronized (this) {
 //...Here you can write your thread safe code...
 }
 }
}
```

#### **45. Can you find whether thread holds lock on object or not?**

**Answer.** holdsLock(object) method can be used to find out whether current thread holds the lock on monitor of specified object.

holdsLock(object) method returns true if the current thread holds the lock on monitor of specified object.

#### **46. . What do you mean by thread starvation?**

**Answer.** When thread does not get enough CPU for its execution **Thread starvation** happens.

**Thread starvation** may happen in following scenarios >

- Low priority threads gets less CPU (time for execution) as compared to high priority threads. **Lower priority thread** may **starve** away waiting to get enough CPU to perform calculations.
- In [deadlock](#) two threads waits for each other to release lock held by them on resources. There both **Threads starves away to get CPU**.
- Thread might be waiting indefinitely for lock on object's monitor (by calling [wait\(\)](#) method), because no other thread is calling [notify\(\)/notifAll\(\)](#) method on object. In that case, **Thread starves** away to get CPU.
- Thread might be waiting indefinitely for lock on object's monitor (by calling [wait\(\)](#) method), but [notify\(\)](#) may be repeatedly awakening some other threads. In that case also **Thread starves** away to get CPU.

#### **47. What is addShutdownHook method in java?**

**Answer.** [addShutdownHook](#) method in java >

- [addShutdownHook](#) method **registers a new virtual-machine shutdown hook**.
- A shutdown hook is a **initialized but unstarted thread**.
- When **JVM starts its shutdown** it will **start all registered shutdown hooks** in some unspecified order and let them run concurrently.

When JVM (Java virtual machine) shuts down >

- When the last non-[daemon](#) thread finishes, or
- when the System.exit is called.

*Once JVM's shutdown has begun new shutdown hook cannot be registered neither previously-registered hook can be de-registered.* Any attempt made to do any of these operations causes an IllegalStateException.

#### **48. How you can handle uncaught runtime exception generated in run method?**

**Answer.** We can use [setDefaultUncaughtExceptionHandler](#) method which can handle uncaught unchecked(runtime) exception generated in run() method.

## What is `setDefaultUncaughtExceptionHandler` method?

`setDefaultUncaughtExceptionHandler` method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.

### *setDefaultUncaughtExceptionHandler method features >*

- **`setDefaultUncaughtExceptionHandler`** method sets the default handler which is called when a thread terminates due to an uncaught unchecked(runtime) exception.
- **`setDefaultUncaughtExceptionHandler`** is a static method method, so we can directly call `Thread.setDefaultUncaughtExceptionHandler` to set the default handler to handle uncaught unchecked(runtime) exception.
- It avoids abrupt termination of thread caused by uncaught runtime exceptions.

### Defining `setDefaultUncaughtExceptionHandler` method >

```
Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler(){
 public void uncaughtException(Thread thread, Throwable throwable)
{
 System.out.println(thread.getName() + " has thrown " +
throwable);
}
});
```

```
class MyRunnable implements Runnable {

 String str;

 /*
 * method will terminate due to an uncaught
unchecked(runtime) exception.
 */
 public void run() {
```

```

 /* String wasn't initialized, so performing any
operation
 * on it will throw NullPointerException and it will caught
by
 * default handler defined in main method.
 */
 str.equals("abc");
 }
}

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class MyClass {
 public static void main(String[] args) {

 Thread thread1 = new Thread(new MyRunnable(),"thread-1");

 /*
 * setDefaultUncaughtExceptionHandler method sets the default
handler
 * which is called when a thread terminates due to an
 * uncaught unchecked(runtime) exception.
 */
 Thread.setDefaultUncaughtExceptionHandler(new
Thread.UncaughtExceptionHandler(){
 public void uncaughtException(Thread thread, Throwable
throwable) {
 System.out.println(thread.getName() + " has thrown " +
throwable);
 }
 });

 thread1.start();
 }
}
/*OUTPUT
thread-1 has thrown java.lang.NullPointerException
*/

```

#### *Output analysis >*

In the above program we have defined **setDefaultUncaughtExceptionHandler** method. And in run method **str** wasn't initialized, calling **str.equals("abc");** threwed **NullPointerException** and it was caught by **default handler defined in main method.**

**49. What is ThreadGroup in java, What is default priority of newly created threadGroup, mention some important ThreadGroup methods ?**

**Answer.** When program starts JVM creates a ThreadGroup named **main**. Unless specified, all newly created threads become members of the **main** thread group.

**ThreadGroup is initialized with default priority of 10.**

*ThreadGroup important methods >*

- **getName()**  
name of ThreadGroup.
- **activeGroupCount()**  
count of active groups in ThreadGroup.
- **activeCount()**  
count of active threads in ThreadGroup.
- **list()**  
list() method has prints ThreadGroups information
- **getMaxPriority()**  
Method returns the maximum priority of ThreadGroup.
- **setMaxPriority(int pri)**  
Sets the maximum priority of ThreadGroup.

**50. What are thread priorities?**

**Answer.**

*Thread Priority range is from 1 to 10.*

Where **1 is minimum priority** and **10 is maximum priority**.

Thread class provides variables of **final static int** type for setting thread priority.

```

/* The minimum priority that a thread can have. */

public final static int MIN_PRIORITY = 1;

/* The default priority that is assigned to a thread. */

public final static int NORM_PRIORITY = 5;

/* The maximum priority that a thread can have. */

public final static int MAX_PRIORITY = 10;

```

Thread with **MAX\_PRIORITY** is likely to get more CPU as compared to low priority threads. But **occasionally low priority thread might get more CPU**. Because thread scheduler schedules thread on discretion of implementation and [thread behaviour is totally unpredictable](#).

Thread with **MIN\_PRIORITY** is likely to get less CPU as compared to high priority threads. But **occasionally high priority thread might less CPU**. Because thread scheduler schedules thread on discretion of implementation and thread behaviour is totally unpredictable.

**setPriority()** method is used for Changing the priority of thread.

**getPriority()** method returns the thread's priority.

## 51. Output question 1.

```

class MyRunnable implements Runnable
{
 public void run(){
 for(int i=0;i<3;i++){
 System.out.println("i="+i+
, ThreadName="+Thread.currentThread().getName());
 }
 }
}

```

```

}

public class MyClass {
 public static void main(String...args){
 MyRunnable runnable=new MyRunnable();
 System.out.println("start main() method");
 Thread thread1=new Thread(runnable);
 Thread thread2=new Thread(runnable);
 thread1.start();
 thread2.start();
 System.out.println("end main() method");
 }
}

```

**Answer.** Thread behaviour is unpredictable because execution of Threads depends on Thread scheduler,

`start main() method` will be the printed first, but after that we cannot guarantee the order of `thread1`, `thread2` and `main` thread they might run simultaneously or sequentially, so order of `end main() method` will not be guaranteed.

```

/*OUTPUT
start main() method
end main() method
i=0 ,ThreadName=Thread-0
i=0 ,ThreadName=Thread-1
i=1 ,ThreadName=Thread-0
i=2 ,ThreadName=Thread-0
i=1 ,ThreadName=Thread-1
i=2 ,ThreadName=Thread-1
*/

```

## 52. Output question 2.

```
class MyRunnable implements Runnable{
 public void run(){
 for(int i=0;i<3;i++){
 System.out.println("i="+i+"
 ,ThreadName='"+Thread.currentThread().getName());
 }
 }
}

public class MyClass {
 public static void main(String...args) throws
InterruptedException{
 System.out.println("In main() method");
 MyRunnable runnable=new MyRunnable();
 Thread thread1=new Thread(runnable);
 Thread thread2=new Thread(runnable);
 thread1.start();
 thread1.join();
 thread2.start();
 thread2.join();
 System.out.println("end main() method");
 }
}
```

**Answer.** We use [\*\*join\(\) method\*\*](#) to ensure all threads that started from main must end in order in which they started and also main should end in last. In other words [\*\*join\(\) method waited for this thread to die.\*\*](#)

```
/*OUTPUT
In main() method
i=0 ,ThreadName=Thread-0
i=1 ,ThreadName=Thread-0
i=2 ,ThreadName=Thread-0
i=0 ,ThreadName=Thread-1
i=1 ,ThreadName=Thread-1
i=2 ,ThreadName=Thread-1
end main() method
*/

```

### 53. Output question 3.

```
class MyRunnable implements Runnable {
 public void run() {
 try {
 while (!Thread.currentThread().isInterrupted()) {
 Thread.sleep(1000);
 System.out.println("x");
 }
 } catch (InterruptedException e) {
 System.out.println(Thread.currentThread().getName()
+ " ENDED");
 }
 }
}
```

```

 }
}

public class MyClass {
 public static void main(String args[]) throws Exception {
 MyRunnable obj = new MyRunnable();
 Thread t = new Thread(obj, "Thread-1");
 t.start();
 System.out.println("press enter");
 System.in.read();
 t.interrupt();
 }
}

```

**Answer.** "press enter" will be printed first then [thread1 will keep on printing x until enter is pressed](#), once enter is pressed "**Thread-1 ENDED**" will be printed.  
**System.in.read()** causes main thread to go from [running to waiting state](#) (thread waits for user input)

```

/* OUTPUT
press enter

x
x
x
x

Thread-1 ENDED
*/

```

#### 54. Output question 4.

```
class MyRunnable implements Runnable{

 public void run(){
 synchronized (this) {
 System.out.println("1 ");
 try {
 this.wait();
 System.out.println("2 ");
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
}

public class MyClass {
 public static void main(String[] args) {
 MyRunnable myRunnable=new MyRunnable();
 Thread thread1=new Thread(myRunnable,"Thread-1");
 thread1.start();

 }
}
```

**Answer.** Thread acquires lock on myRunnable object so **1** was printed but notify wasn't called so **2** will never be printed, this is called frozen process. Deadlock is formed, these type of **deadlocks** are called **Frozen processes**.

```
/*OUTPUT
1
*/
```

### 55. Output question 5.

```
import java.util.ArrayList;
/* Producer is producing, Producer will allow consumer to
 * consume only when 10 products have been produced (i.e. when
production is over).
*/
class Producer implements Runnable{
 ArrayList<Integer> sharedQueue;
 Producer(){
 sharedQueue=new ArrayList<Integer>();
 }
 @Override
 public void run(){
 synchronized (this) {
 for(int i=1;i<=3;i++){ //Producer will produce 10
products
 sharedQueue.add(i);
 System.out.println("Producer is still
Producing, Produced : "+i);

 try{

```

```
 Thread.sleep(1000);

 }catch(InterruptedException
e){e.printStackTrace();}

}

System.out.println("Production is over, consumer can
consume.");
this.notify();
}

}

}

}

class Consumer extends Thread{
Producer prod;

Consumer(Producer obj){
prod=obj;
}

public void run(){
synchronized (this.prod) {

System.out.println("Consumer waiting for production
to get over.");
try{
this.prod.wait();
}catch(InterruptedException
e){e.printStackTrace();}
}
}
```

```
}

 int productSize=this.prod.sharedQueue.size();
 for(int i=0;i<productSize;i++)
 System.out.println("Consumed : "+
this.prod.sharedQueue.remove(0) +" ");

}

}

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

 Producer prod=new Producer();
 Consumer cons=new Consumer(prod);

 Thread prodThread=new Thread(prod, "prodThread");
 Thread consThread=new Thread(cons, "consThread");

 consThread.start();
 Thread.sleep(100); //minor delay.
 prodThread.start();

}
}
```

```
}
```

**Answer.** Because of minor delay delay consThread surely started before producer thread. "Consumer waiting for production to get over." printed first than producer produced than "Production is over, consumer can consume." than consumer consumed.

The above program is classical example of [how to solve Consumer Producer problem by using wait\(\) and notify\(\) methods.](#)

```
/*OUTPUT
Consumer waiting for production to get over.
Producer is still Producing, Produced : 1
Producer is still Producing, Produced : 2
Producer is still Producing, Produced : 3
Production is over, consumer can consume.
Consumed : 1
Consumed : 2
Consumed : 3
*/
```

## 56. Output question 6.

```
class MyRunnable implements Runnable{

 public void run(){
```

```

synchronized (this) {
 System.out.print("1 ");
 try {
 this.wait(1000);
 System.out.print("2");
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
}
}

public class MyClass {
 public static void main(String[] args) {
 MyRunnable myRunnable=new MyRunnable();
 Thread thread1=new Thread(myRunnable,"Thread-1");
 thread1.start();
 }
}

```

**Answer.** First 1 will be printed then even if [notify\(\) or notifyAll\(\)](#) is not called, thread will be [notified after 1000 millisec](#) and 2 will be printed.

```

/*OUTPUT
1 2
*/

```

### 57. Output question 7.

```

class MyRunnable implements Runnable {
 public void run() {

```

```
System.out.println(Thread.currentThread().getName() + " has
started");

 try {

 Thread.sleep(100); //ensure that main thread don't
complete before Thread-1

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 System.out.println(Thread.currentThread().getName() + " has
ended");

}

}

public class MyClass {

 public static void main(String... args) throws
InterruptedException {

 System.out.println(Thread.currentThread().getName() + " has
started");

 Thread thread1 = new Thread(new MyRunnable(), "Thread-1");
 thread1.start();
 thread1.sleep(10000);

 System.out.println(Thread.currentThread().getName() + " has
ended");

 }

}

/*OUTPUT

main has started
Thread-1 has started
Thread-1 has ended
main has ended
```

```
 */
```

### Answer.

sleep() is a static method, causes the currently executing thread to sleep for the specified number of milliseconds.

Calling thread1.sleep(1000); will show warning - The static method sleep(long) from the type Thread should be accessed in a static way.

In the program first main thread started, than it invoked Thread-1, then Thread-1 called sleep(100) method to ensure that main thread don't complete before Thread-1, than execution control went to main thread and it called thread1.sleep(1000) **but rather than putting Thread-1 on sleep it made main thread to sleep.** And Thread-1 ended before main thread.

### 58. Output question 8.

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 synchronized (this) {
 try{
 System.out.print("2 ");
 Thread.sleep(1000);
 }catch(InterruptedException e){e.printStackTrace();}
 this.notify();
 System.out.print("3 ");
 }
 }
}

class MyRunnable2 extends Thread{
 MyRunnable1 prod;
```

```
MyRunnable2(MyRunnable1 obj){
 prod=obj;
}

public void run(){
 synchronized (this.prod) {

 System.out.print("1 ");
 try{
 this.prod.wait();
 }catch(InterruptedException
e){e.printStackTrace();}

 }

 System.out.print("4 ");

}

}

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

 MyRunnable1 myRunnable1=new MyRunnable1();
 MyRunnable2 myRunnable2=new MyRunnable2(myRunnable1);
```

```

Thread thread1=new Thread(myRunnable1, "Thread-1");
Thread thread2=new Thread(myRunnable2, "Thread-2");

 thread2.start();
 Thread.sleep(100); //This minor delay will ensure that
Thread-1 thread starts Thread-2
 thread1.start();

}

}

```

**Answer.** [Wait\(\)](#) method causes the current thread to wait until another thread invokes the notify() or notifyAll() method for this object.

Now, as soon as [notify\(\) or notifyall\(\)](#) method is called it **notifies the waiting thread, but object monitor is not yet available. Object monitor is available only when thread exits synchronized block or synchronized method.** So, what happens is code after notify() is also executed and execution is done until we reach end of synchronized block.

**The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object**

```

/*OUTPUT
1 2 3 4
*/

```

## 59. Output question 9.

```

class MyThread extends Thread {
 MyThread() {

```

```

 System.out.print("1 ");
 }
 public void run() {
 System.out.print("2 ");
 }
}
public class MyClass {
 public static void main(String[] args) {
 Thread thread1 = new MyThread() {
 public void run() {
 System.out.print("3 ");
 }
 };
 thread1.start();
 }
}

```

**Answer.**

`new MyThread()` > created instance of an anonymous inner class.  
 constructor was called which printed 1  
 than overridden run() method of anonymous inner class was invoked, which printed 3.  
 /\*OUTPUT  
 1 3  
 \*/

**60. Output question 10.**

```

class MyRunnable implements Runnable{
 public void run(){
 method();
 }
 synchronized void method(){
 for(int i=0;i<2;i++){
 System.out.println(Thread.currentThread().getName());
 }
 }
}

```

```

}

public class MyClass {
 public static void main(String...args){
 MyRunnable runnable=new MyRunnable();
 Thread thread1=new Thread(runnable,"Thread-1");
 Thread thread2=new Thread(runnable,"Thread-2");
 thread1.start();
 thread2.start();
 }
}

//q6

```

**Answer.** Thread behavior is unpredictable because execution of Threads depends on Thread scheduler, either of thread1 and thread2 can start first and synchronized method will be executed by one thread at a time.

```

/*OUTPUT if Thread-1 entered first in synchronized block.

Thread-1
Thread-1
Thread-2
Thread-2
*/
/*OUTPUT if Thread-2 entered first in synchronized block.

Thread-2
Thread-2
Thread-1
Thread-1

```

\*/

## 61. Output question 11.

```
public class MyClass {
 public static void main(String[] args) {
 Thread thread1=new Thread("Thread-1"){
 public void run() {

 synchronized (String.class) {

 try {
 Thread.sleep(100);
 } catch (InterruptedException e)
{e.printStackTrace();}

 System.out.println("1 ");
 synchronized (Object.class) {
 System.out.println("2 ");
 }
 }

 System.out.println("3 ");
 }
 };
 Thread thread2=new Thread("Thread-2"){
 public void run() {
 }
 }
}
```

```

synchronized (Object.class) {
 System.out.println("4 ");

 try {
 Thread.sleep(100);
 } catch (InterruptedException e)
{e.printStackTrace();}

synchronized (String.class) {
 System.out.println("5 ");

}

System.out.println("6 ");
}

};

thread1.start();
thread2.start();
}

}

```

**Answer.** Deadlock is formed in above program :

**Thread-1 acquires lock on String.class** and then calls sleep() method which gives Thread-2 the chance to execute immediately after Thread-1 has acquired lock on String.class and **Thread-2 acquires lock on Object.class** then calls sleep() method and **now it waits for Thread-1 to release lock on String.class**.

**Conclusion:**

Now, **Thread-1 is waiting for Thread-2 to release lock on Object.class and Thread-2 is waiting for Thread-1 to release lock on String.class** and deadlock is formed.

```
/*OUTPUT
4
1
*/
```

## 62. Output question 12.

```
public class MyClass {
 public static void main(String[] args) throws InterruptedException {
 synchronized (args) {
 System.out.print("1 ");
 args.wait();
 System.out.print("2 ");
 }
 }
}
```

**Answer.** Though this question looks bit similar to output question 4 but intention is to show args is object and we can acquire lock on it.

Thread acquires lock on args object but notify wasn't called so 2 will never be printed, this is called frozen process.

```
/*OUTPUT
1
```

\*/

### 63. Output question 13.

```
package o13_k15;

class Class2 {

 void method2(String name) {
 for (int x = 1; x <=2; x++) {
 System.out.println(Thread.currentThread().getName());
 }
 }

 public class MyClass implements Runnable {
 Class2 obj2;
 public static void main(String[] args) {
 new MyClass().method1();
 }
 void method1() {
 obj2 = new Class2();
 new Thread(new MyClass()).start();
 new Thread(new MyClass()).start();
 }
 public void run() {
 obj2.method2(Thread.currentThread().getName());
 }
 }
}
```

**Answer.** Program will face NullPointerException at Class2 `obj2`, we must make it static. As `new Thread(new MyClass()).start();` creates thread on new instance of MyClass.

If Class2 `obj2` is made static, than

`Thread-0` and `Thread-1` will be printed twice but in unpredictable order.

So, output will be different in subsequent executions,(as shown below)-

```
/*OUTPUT
Thread-1
Thread-1
Thread-0
Thread-0
*/
/*OUTPUT
Thread-0
Thread-1
Thread-1
Thread-0
*/
```

#### 64. Output question 14.

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
```

```
 method1();

 else

 method2();

}

synchronized void method1(){

 System.out.println(Thread.currentThread().getName()

 +" in synchronized void method1() started");

 try {

 Thread.sleep(2000);

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 System.out.println(Thread.currentThread().getName()

 +" in synchronized void method1() ended");

}

synchronized void method2(){

 System.out.println(Thread.currentThread().getName()

 +" in synchronized void method2() started");

 try {

 Thread.sleep(2000);

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 System.out.println(Thread.currentThread().getName()+

 " in synchronized void method2() ended");

}

}

public class MyClass {
```

```

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

 MyRunnable1 myRunnable1=new MyRunnable1();

 Thread thread1=new Thread(myRunnable1,"Thread-1");
 Thread thread2=new Thread(myRunnable1,"Thread-2");
 thread1.start();
 Thread.sleep(10);//Just to ensure Thread-1 starts before Thread-2
 thread2.start();

}

}

```

**Answer.** Here when Thread-1 is in **synchronized method1()** it must be **holding lock on object's monitor** and will release lock on object's monitor only when it exits **synchronized method1()**. So, Thread-2 will have to **wait** for Thread-1 to release lock on object's monitor so that it could enter **synchronized method2()**.

**Likewise**, Thread-2 even cannot enter **synchronized method1()** which is being executed by Thread-1. Thread-2 will have to **wait** for Thread-1 to release lock on object's monitor so that it could enter **synchronized method1()**.

```

/*OUTPUT

Thread-1 in synchronized void method1() started
Thread-1 in synchronized void method1() ended
Thread-2 in synchronized void method2() started
Thread-2 in synchronized void method2() ended
*/

```

## 65. Output question 15.

```
class MyRunnable implements Runnable{

 public void run(){

 System.out.println("1 ");
 try {
 this.wait();
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println("2 ");

 }
}

public class WaitNoParaMethod {
 public static void main(String[] args) {
 MyRunnable myRunnable=new MyRunnable();
 Thread thread1=new Thread(myRunnable, "Thread-1");
 thread1.start();

 }
}
```

## Answer.

**IllegalMonitorStateException** is thrown at runtime, as [wait\(\)](#) method was called without acquiring lock on object monitor.

```
/*OUTPUT
1
Exception in thread "Thread-1" java.lang.IllegalMonitorStateException
 at java.lang.Object.wait(Native Method)
 at java.lang.Object.wait(Object.java:503)
 at o15_wait_IllegalMoni.MyRunnable.run(WaitNoParaMethod.java:9)
 at java.lang.Thread.run(Unknown Source)
*/

```

## 66. Output question 16.

```
public class MyClass implements Runnable{

 @Override
 public void run() {
 System.out.println("1");
 }

 public static void main(String[] args) {
 MyClass obj=new MyClass();
 Thread thread1=new Thread(obj, "Thread-1");
 thread1.start();
 thread1.start();
 }
}
```

**Answer.** [we cannot start Thread again](#), doing so will throw runtimeException `java.lang.IllegalThreadStateException`. The reason is once run() method is executed by Thread, it goes into [dead state](#).

Let's take an example-

Thinking of starting thread again and calling start() method on it (which internally is going to call run() method) for us is some what like asking dead man to wake up and run. As, after completing his life person goes to **dead state**.

```
/*OUTPUT
1
Exception in thread "main" java.lang.IllegalThreadStateException
at java.lang.Thread.start(Unknown Source)
*/
```

## 67. Output question 17.

```
class MyThread extends Thread {
}

public class MyClass {
 public static void main(String[] args) {
 Thread thread1=new MyThread();
 thread1.start();
 }
}
```

**Answer.** Nothing will be printed in output.

**When we call start() method on thread, it internally calls run() method with newly created thread. So, if we don't override run() method newly created thread won't be called and nothing will happen.**

### 68. Output question 18.

```
class MyThread extends Thread {
 public void run() {
 System.out.println("1");
 }
 public void start(){
 System.out.println("2");
 }
}

public class MyClass {
 public static void main(String[] args) {
 MyThread thread1=new MyThread();
 thread1.start();
 }
}
```

**Answer.** When we call start() method on thread, it internally calls run() method with newly created thread. So, if we override start() method, run() method will not be called until we write code for calling run() method.

```
/*OUTPUT
2
*/
```

## 69. Output question 19.

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
 method1();
 else
 method2();
 }
 static synchronized void method1(){
 System.out.println(Thread.currentThread().getName() +
 " in synchronized void method1() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method1() ended");
 }

 static synchronized void method2(){
 }
```

```

System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() started");

try {
 Thread.sleep(2000);
} catch (InterruptedException e) {
 e.printStackTrace();
}

System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() ended");
}

}

public class MyClass {
 public static void main(String args[]) throws InterruptedException{
 MyRunnable1 object1=new MyRunnable1();
 MyRunnable1 object2=new MyRunnable1();

 Thread thread1=new Thread(object1,"Thread-1");
 Thread thread2=new Thread(object2,"Thread-2");
 thread1.start();
 Thread.sleep(10); //Just to ensure Thread-1 starts before Thread-2
 thread2.start();
 }
}

```

**Answer.** It might confuse you a bit that threads are created on different objects. But, not to forgot that **multiple objects may exist but there is always one class's class object lock available.**

Here, when Thread-1 is in **static synchronized method1()** it must be **holding lock on class class's object** and will release lock on class's class object only when it exits **static synchronized method1()**. So, Thread-2 will have to wait for Thread-1 to release lock on class's class object so that it could enter **static synchronized method2()**.

```
/*OUTPUT
Thread-1 in synchronized void method1() started
Thread-1 in synchronized void method1() ended
Thread-2 in synchronized void method2() started
Thread-2 in synchronized void method2() ended
*/
```

## 70. Output question 20.

```
class MyRunnable1 implements Runnable{
 @Override
 public void run(){
 if(Thread.currentThread().getName().equals("Thread-1"))
 method1();
 else
 method2();
 }
 synchronized void method1(){
 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method1() started");
 try {
 Thread.sleep(2000);
 } catch (InterruptedException e) {
```

```
 e.printStackTrace();
 }

 System.out.println(Thread.currentThread().getName()+
 " in synchronized void method1() ended");

}

synchronized void method2(){

 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() started");

 try {

 Thread.sleep(2000);

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 System.out.println(Thread.currentThread().getName()
 +" in synchronized void method2() ended");

}

}

public class MyClass {

 public static void main(String args[]) throws InterruptedException{
 MyRunnable1 object1=new MyRunnable1();
 MyRunnable1 object2=new MyRunnable1();

 Thread thread1=new Thread(object1,"Thread-1");
 Thread thread2=new Thread(object2,"Thread-2");
 thread1.start();
 Thread.sleep(10); //Just to ensure Thread-1 starts before Thread-2
 thread2.start();
 }
}
```

```

 }
}

```

**Answer.** Here when Thread-1 is in **synchronized method1()** it must be **holding lock on object1's monitor**. Thread-2 will acquire lock on **object2's monitor** and enter **synchronized method2()**.

```

/*OUTPUT
Thread-1 in synchronized void method1() started
Thread-2 in synchronized void method2() started
Thread-1 in synchronized void method1() ended
Thread-2 in synchronized void method2() ended
*/

```

## 71. Output question 21.

```

public class MyClass extends Thread{
 public void run() {
 method1();
 }

 public static void method1() {
 synchronized (this) {
 System.out.println("2 ");
 }
 }
}

```

```

 }
}

public static void main(String[] args) {
 new Thread(new MyClass()).start();
}

}

```

**Answer.** We will face compilation error at line `synchronized (this)` can't use in static context, because it's not possible to obtain lock on object from static method. Though we can obtain lock on class's class object, so `synchronized (MyClass.class)` will be a valid statement.

## 72. Question 82. Output question 22.

```

public class MyClass {

 public static void main(String[] args) {
 System.out.println("1 ");
 InnerClass i=new InnerClass();
 i.start();
 System.out.println("2 ");
 }

 static class InnerClass extends Thread{
 public void run()throws RuntimeException{
 throw new RuntimeException();
 }
 }
}

```

```
}
```

**Answer.** Program will compile as run() method can throw RuntimeException. 1 & 2 will be present in output and will throw `java.lang.RuntimeException` at runtime.

```
/*OUTPUT
1
2
Exception in thread "Thread-0" java.lang.RuntimeException
 at o22.s$InnerClass.run(s.java:13)
*/

```

### 73. Deadlock in Java Multithreading

**synchronized** keyword is used to make the class or method thread-safe which means only one thread can have lock of synchronized method and use it, other threads have to wait till the lock releases and anyone of them acquire that lock. It is important to use if our program is running in multi-threaded environment where two or more threads execute simultaneously. But sometimes it also causes a problem which is called **Deadlock**. Below is a simple example of Deadlock condition.

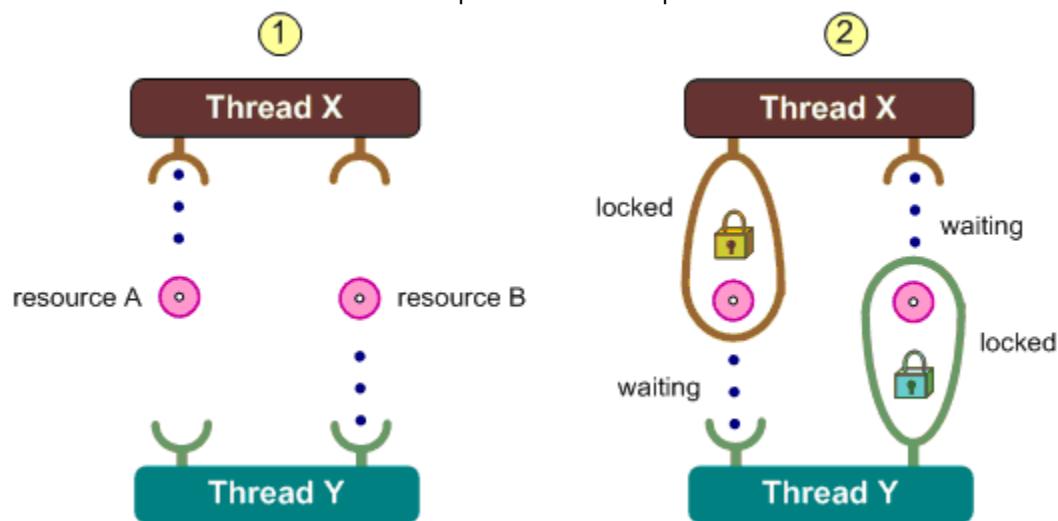


Image source: <https://software.intel.com/en-us/articles/multi-threading-in-the-net-environment>

```
// Java program to illustrate Deadlock
// in multithreading.

class Util
{
 .
 // Util class to sleep a thread
 static void sleep(long millis)
 {
 try
 {
 Thread.sleep(millis);
 }
 catch (InterruptedException e)
 {
 e.printStackTrace();
 }
 }

 }

// This class is shared by both threads

class Shared
{
 // first synchronized method
 synchronized void test1(Shared s2)
 {
 System.out.println("test1-begin");
 Util.sleep(1000);

 // taking object lock of s2 enters
 }
}
```

```
// into test2 method
s2.test2(this);
System.out.println("test1-end");
}

// second synchronized method
synchronized void test2(Shared s1)
{
 System.out.println("test2-begin");
 Util.sleep(1000);

 // taking object lock of s1 enters
 // into test1 method
 s1.test1(this);
 System.out.println("test2-end");
}

}

class Thread1 extends Thread
{
 private Shared s1;
 private Shared s2;

 // constructor to initialize fields
 public Thread1(Shared s1, Shared s2)
 {
 this.s1 = s1;
```

```
 this.s2 = s2;

 }

 // run method to start a thread
 @Override
 public void run()
 {
 // taking object lock of s1 enters
 // into test1 method
 s1.test1(s2);
 }
}

class Thread2 extends Thread
{
 private Shared s1;
 private Shared s2;

 // constructor to initialize fields
 public Thread2(Shared s1, Shared s2)
 {
 this.s1 = s1;
 this.s2 = s2;
 }

 // run method to start a thread
 @Override
```

```
public void run()
{
 // taking object lock of s2
 // enters into test2 method
 s2.test2(s1);
}

}

public class GFG
{
 public static void main(String[] args)
 {
 // creating one object
 Shared s1 = new Shared();

 // creating second object
 Shared s2 = new Shared();

 // creating first thread and starting it
 Thread1 t1 = new Thread1(s1, s2);
 t1.start();

 // creating second thread and starting it
 Thread2 t2 = new Thread2(s1, s2);
 t2.start();

 // sleeping main thread
 }
}
```

```

 Util.sleep(2000);
 }
}

```

Run on IDE

```

Output : test1-begin
test2-begin

```

It is not recommended to run the above program with online IDE. We can copy the source code and run it on our local machine. We can see that it runs for indefinite time, because threads are in deadlock condition and doesn't let code to execute. Now let's see step by step what is happening there.

1. Thread t1 starts and calls test1 method by taking the object lock of s1.
2. Thread t2 starts and calls test2 method by taking the object lock of s2.
3. t1 prints test1-begin and t2 prints test2 begin and both waits for 1 second, so that both threads can be started if any of them is not.
4. t1 tries to take object lock of s2 and call method test2 but as it is already acquired by t2 so it waits till it becomes free. It will not release lock of s1 until it gets lock of s2.
5. Same happens with t2. It tries to take object lock of s1 and call method test1 but it is already acquired by t1, so it has to wait till t1 releases the lock. t2 will also not release lock of s2 until it gets lock of s1.
6. Now, both threads are in wait state, waiting for each other to release locks. Now there is a race around condition that who will release the lock first.
7. As none of them is ready to release lock, so this is the Dead Lock condition.
8. When you will run this program, it will look like execution is paused.

#### **Detect Dead Lock condition**

We can also detect deadlock by running this program on cmd. We have to collect Thread Dump. Command to collect depends on OS type. If we are using Windows and Java 8, command is jcmd \$PID Thread.print

We can get PID by running jps command. Thread dump for above program is below:

```

5524:
2017-04-21 09:57:39
Full thread dump Java HotSpot(TM) 64-Bit Server VM (25.25-b02 mixed mode):

"DestroyJavaVM" #12 prio=5 os_prio=0 tid=0x0000000002690800 nid=0xba8 waiting on condition
[0x0000000000000000]
 java.lang.Thread.State: RUNNABLE

```

```

"Thread-1" #11 prio=5 os_prio=0 tid=0x0000000018bbf800 nid=0x12bc waiting for monitor entry
[0x000000001937f000]

 java.lang.Thread.State: BLOCKED (on object monitor)
 at Shared.test1(GFG.java:15)
 - waiting to lock (a Shared)
 at Shared.test2(GFG.java:29)
 - locked (a Shared)
 at Thread2.run(GFG.java:68)

"Thread-0" #10 prio=5 os_prio=0 tid=0x0000000018bbc000 nid=0x1d8 waiting for monitor entry
[0x000000001927f000]

 java.lang.Thread.State: BLOCKED (on object monitor)
 at Shared.test2(GFG.java:25)
 - waiting to lock (a Shared)
 at Shared.test1(GFG.java:19)
 - locked (a Shared)
 at Thread1.run(GFG.java:49)

"Service Thread" #9 daemon prio=9 os_prio=0 tid=0x000000001737d800 nid=0x1680 runnable
[0x0000000000000000]

 java.lang.Thread.State: RUNNABLE

"C1 CompilerThread2" #8 daemon prio=9 os_prio=2 tid=0x000000001732b800 nid=0x17b0 waiting on
condition [0x0000000000000000]

 java.lang.Thread.State: RUNNABLE

"C2 CompilerThread1" #7 daemon prio=9 os_prio=2 tid=0x0000000017320800 nid=0x7b4 waiting on
condition [0x0000000000000000]

 java.lang.Thread.State: RUNNABLE

"C2 CompilerThread0" #6 daemon prio=9 os_prio=2 tid=0x000000001731b000 nid=0x21b0 waiting on
condition [0x0000000000000000]

 java.lang.Thread.State: RUNNABLE

"Attach Listener" #5 daemon prio=5 os_prio=2 tid=0x0000000017319800 nid=0x1294 waiting on
condition [0x0000000000000000]

 java.lang.Thread.State: RUNNABLE

```

```

"Signal Dispatcher" #4 daemon prio=9 os_prio=2 tid=0x0000000017318000 nid=0x1efc runnable
[0x0000000000000000]

 java.lang.Thread.State: RUNNABLE

"Finalizer" #3 daemon prio=8 os_prio=1 tid=0x0000000002781800 nid=0x5a0 in Object.wait()
[0x000000001867f000]

 java.lang.Thread.State: WAITING (on object monitor)
 at java.lang.Object.wait(Native Method)
 - waiting on (a java.lang.ref.ReferenceQueue$Lock)
 at java.lang.ref.ReferenceQueue.remove(Unknown Source)
 - locked (a java.lang.ref.ReferenceQueue$Lock)
 at java.lang.ref.ReferenceQueue.remove(Unknown Source)
 at java.lang.ref.Finalizer$FinalizerThread.run(Unknown Source)

"Reference Handler" #2 daemon prio=10 os_prio=2 tid=0x000000000277a800 nid=0x15b4 in
Object.wait() [0x000000001857f000]

 java.lang.Thread.State: WAITING (on object monitor)
 at java.lang.Object.wait(Native Method)
 - waiting on (a java.lang.ref.Reference$Lock)
 at java.lang.Object.wait(Unknown Source)
 at java.lang.ref.Reference$ReferenceHandler.run(Unknown Source)
 - locked (a java.lang.ref.Reference$Lock)

"VM Thread" os_prio=2 tid=0x00000000172e6000 nid=0x1fec runnable

"GC task thread#0 (ParallelGC)" os_prio=0 tid=0x00000000026a6000 nid=0x21fc runnable

"GC task thread#1 (ParallelGC)" os_prio=0 tid=0x00000000026a7800 nid=0x2110 runnable

"GC task thread#2 (ParallelGC)" os_prio=0 tid=0x00000000026a9000 nid=0xc54 runnable

"GC task thread#3 (ParallelGC)" os_prio=0 tid=0x00000000026ab800 nid=0x704 runnable

"VM Periodic Task Thread" os_prio=2 tid=0x0000000018ba0800 nid=0x610 waiting on condition

JNI global references: 6

```

```

Found one Java-level deadlock:
=====
"Thread-1":
 waiting to lock monitor 0x00000000018bc1e88 (object 0x00000000d5d645a0, a Shared),
 which is held by "Thread-0"

"Thread-0":
 waiting to lock monitor 0x00000000002780e88 (object 0x00000000d5d645b0, a Shared),
 which is held by "Thread-1"

Java stack information for the threads listed above:
=====
"Thread-1":
 at Shared.test1(GFG.java:15)
 - waiting to lock (a Shared)
 at Shared.test2(GFG.java:29)
 - locked (a Shared)
 at Thread2.run(GFG.java:68)

"Thread-0":
 at Shared.test2(GFG.java:25)
 - waiting to lock (a Shared)
 at Shared.test1(GFG.java:19)
 - locked (a Shared)
 at Thread1.run(GFG.java:49)

Found 1 deadlock.

```

As we can see there is clearly mentioned that found 1 deadlock. It is possible that the same message appears when you try on your machine.

### Avoid Dead Lock condition

We can avoid dead lock condition by knowing its possibilities. It's a very complex process and not easy to catch. But still if we try, we can avoid this. There are some methods by which we can avoid this condition. We can't completely remove its possibility but we can reduce.

- **Avoid Nested Locks :** This is the main reason for dead lock. Dead Lock mainly happens when we give locks to multiple threads. Avoid giving lock to multiple threads if we already have given to one.
- **Avoid Unnecessary Locks :** We should have lock only those members which are required. Having lock on unnecessarily can lead to dead lock.
- **Using thread join :** Dead lock condition appears when one thread is waiting other to finish. If this condition occurs we can use Thread.join with maximum time you think the execution will take.

#### **Important Points :**

- If threads are waiting for each other to finish, then the condition is known as Deadlock.
- Deadlock condition is a complex condition which occurs only in case of multiple threads.
- Deadlock condition can break our code at run time and can destroy business logic.
- We should avoid this condition as much as we can.

## Concurrency

### 1. What is ThreadPool?

ThreadPool is a pool of threads which **reuses a fixed number of threads** to execute tasks.

At any point, **at most nThreads threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available.**

ThreadPool implementation internally uses [LinkedBlockingQueue](#) for adding and removing tasks.

In this post i will be using LinkedBlockingQueue provided by java Api, you can refer this post for [implementing ThreadPool using custom LinkedBlockingQueue](#).

We may use [Executor and ExecutorService framework in java](#) for managing thread life cycle.

### 2. What is ThreadFactory? Why is it implemented?

An object that creates new threads on demand. Using thread factories removes hardwiring of calls to [new Thread](#), enabling applications to use special thread subclasses, priorities, etc.

The simplest implementation of this interface is just:

```
class SimpleThreadFactory implements ThreadFactory {
 public Thread newThread(Runnable r) {
 return new Thread(r);
 }
}
```

```
}
```

The [Executors.defaultThreadFactory\(\)](#) method provides a more useful simple implementation, that sets the created thread context to known values before returning it.

**Since:**

1.5

### 3. What is ThreadGroup? Why is it used?

Java provides a convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method call.

Java thread group is implemented by *java.lang.ThreadGroup* class.

| No. | Constructor                                  | Description                                              |
|-----|----------------------------------------------|----------------------------------------------------------|
| 1)  | ThreadGroup(String name)                     | creates a thread group with given name.                  |
| 2)  | ThreadGroup(ThreadGroup parent, String name) | creates a thread group with given parent group and name. |

Important methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of important methods are given below.

| No. | Method                 | Description                                         |
|-----|------------------------|-----------------------------------------------------|
| 1)  | int activeCount()      | returns no. of threads running in current group.    |
| 2)  | int activeGroupCount() | returns a no. of active group in this thread group. |
| 3)  | void destroy()         | destroys this thread group and all its sub groups.  |
| 4)  | String getName()       | returns the name of this group.                     |

|    |                            |                                                       |
|----|----------------------------|-------------------------------------------------------|
| 5) | ThreadGroup<br>getParent() | returns the parent of this group.                     |
| 6) | void interrupt()           | interrupts all threads of this group.                 |
| 7) | void list()                | prints information of this group to standard console. |

Let's see a code to group multiple threads.

1. ThreadGroup tg1 = **new** ThreadGroup("Group A");
2. Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");
3. Thread t2 = **new** Thread(tg1,**new** MyRunnable(),"two");
4. Thread t3 = **new** Thread(tg1,**new** MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

1. Thread.currentThread().getThreadGroup().interrupt();

#### **4. Explain about the shutDownHook in java.**

The shutdown hook can be used to perform cleanup resource or save the state when JVM shuts down normally or abruptly. Performing clean resource means closing log file, sending some alerts or something else. So if you want to execute some code before JVM shuts down, use shutdown hook.

#### **When does the JVM shut down?**

The JVM shuts down when:

- o user presses ctrl+c on the command prompt
- o System.exit(int) method is invoked
- o user logoff
- o user shutdown etc.

---

#### **The addShutdownHook(Thread hook) method**

The addShutdownHook() method of Runtime class is used to register the thread with the Virtual Machine. Syntax:

1. **public void** addShutdownHook(Thread hook){}

The object of Runtime class can be obtained by calling the static factory method getRuntime(). For example:

```
Runtime r = Runtime.getRuntime();
```

#### **Factory method**

The method that returns the instance of a class is known as factory method.

### **Simple example of Shutdown Hook**

```

1. class MyThread extends Thread{
2. public void run(){
3. System.out.println("shut down hook task completed..");
4. }
5. }
6.
7. public class TestShutdown1{
8. public static void main(String[] args)throws Exception {
9.
10. Runtime r=Runtime.getRuntime();
11. r.addShutdownHook(new MyThread());
12.
13. System.out.println("Now main sleeping... press ctrl+c to exit");
14. try{Thread.sleep(3000);} catch (Exception e) {}
15. }
16. }
```

```
Output:Now main sleeping... press ctrl+c to exit
 shut down hook task completed..
```

### **5. Difference between call() and the run() methods?**

Callable needs to implement `call()` method while a Runnable needs to implement `run()` method.

A Callable can return a value but a Runnable cannot.

A **Callable** can throw checked exception but a **Runnable** cannot.  
A **Callable** can be used with **ExecutorService#invokeXXX** methods but a **Runnable** cannot be.

```
public interface Runnable {
 void run();
}

public interface Callable<V> {
 V call() throws Exception;
}
```

## 6. What is Java.util.concurrent.CyclicBarrier ?

cyclicBarrier is used to make threads wait for each other. It is used when different threads process a part of computation and when all threads have completed the execution, the result needs to be combined in the parent thread. In other words, a CyclicBarrier is used when multiple thread carry out different sub tasks and the output of these sub tasks need to be combined to form the final output. After completing its execution, threads call await() method and wait for other threads to reach the barrier. Once all the threads have reached, the barriers then give the way for threads to proceed.

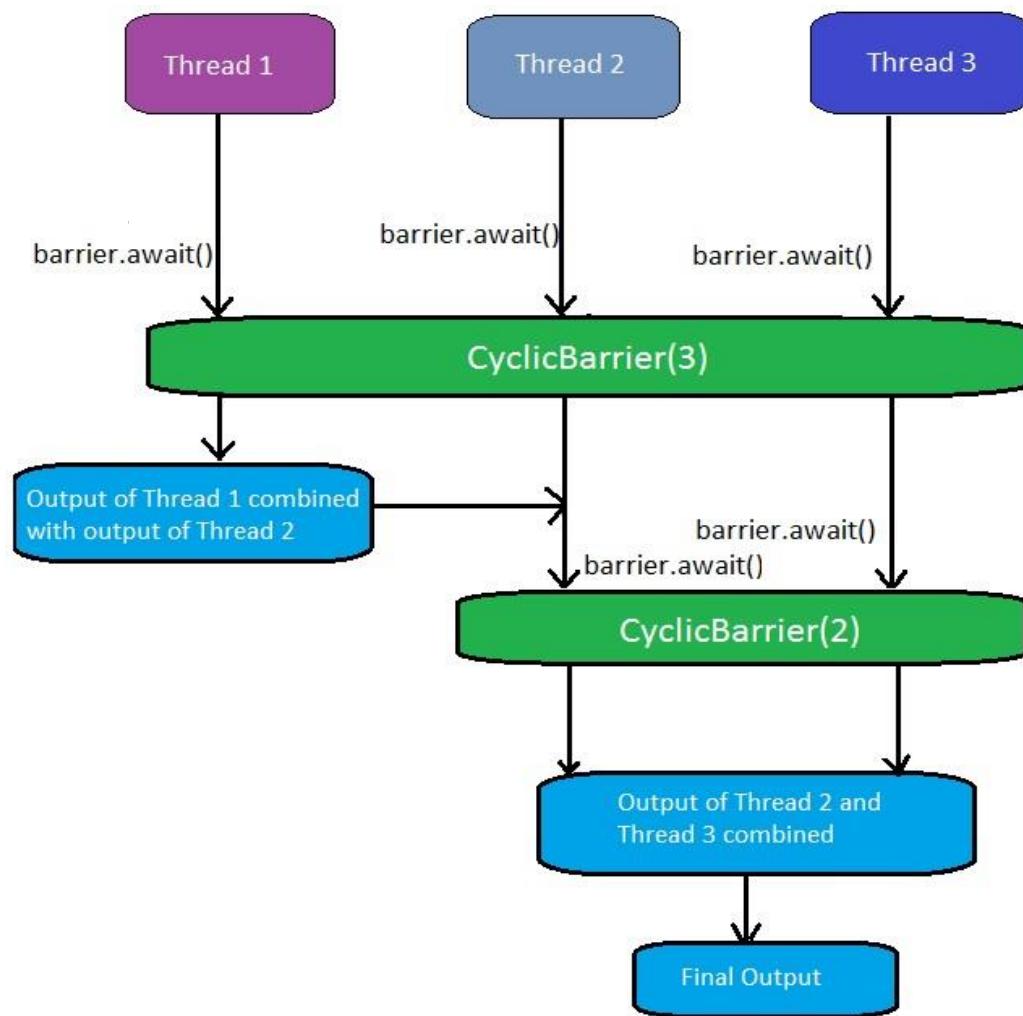
### **Working of CyclicBarrier**

CyclicBarriers are defined in `java.util.concurrent` package. First a new instance of a CyclicBarriers is created specifying the number of threads that the barriers should wait upon.

```
CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads);
```

Each and every thread does some computation and after completing it's execution, calls await() methods as shown:

```
public void run()
{
 // thread does the computation
 newBarrier.await();
}
```



Once the number of threads that called await() equals **numberOfThreads**, the barrier then gives a way for the waiting threads. The CyclicBarrier can also be initialized with some action that is performed once all the threads have reached the barrier. This action can combine/utilize the result of computation of individual thread waiting in the barrier.

```

Runnable action = ...
//action to be performed when all threads reach the barrier;
CyclicBarrier newBarrier = new CyclicBarrier(numberOfThreads,
action);

```

### Important Methods of CyclicBarrier:

1. **getParties:** Returns the number of parties required to trip this barrier.  
**Syntax:**

```
public int getParties()
```

**Returns:**

the number of parties required to trip this barrier

2. **reset:** Resets the barrier to its initial state.

**Syntax:**

```
public void reset()
```

**Returns:**

void but resets the barrier to its initial state. If any parties are currently waiting at the barrier, they will return with a BrokenBarrierException.

3. **isBroken:** Queries if this barrier is in a broken state.

**Syntax:**

```
public boolean isBroken()
```

**Returns:**

true if one or more parties broke out of this barrier due to interruption or timeout since construction or the last reset, or a barrier action failed due to an exception; false otherwise.

4. **getNumberWaiting:** Returns the number of parties currently waiting at the barrier.

**Syntax:**

```
public int getNumberWaiting()
```

**Returns:**

the number of parties currently blocked in await()

5. **await:** Waits until all parties have invoked await on this barrier.

**Syntax:**

```
public int await() throws InterruptedException,
BrokenBarrierException
```

**Returns:**

the arrival index of the current thread, where index getParties() – 1 indicates the first to arrive and zero indicates the last to arrive.

6. **await:** Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.

**Syntax:**

7. `public int await(long timeout, TimeUnit unit)`
8. `throws InterruptedException,`  
`BrokenBarrierException, TimeoutException`

**Returns:**

the arrival index of the current thread, where `index getParties() – 1` indicates the first to arrive and zero indicates the last to arrive

```
//JAVA program to demonstrate execution on Cyclic Barrier

import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

class Computation1 implements Runnable
{
 public static int product = 0;
 public void run()
 {
 product = 2 * 3;
 try
 {
 Tester.newBarrier.await();
 }
 catch (InterruptedException | BrokenBarrierException e)
 {
 e.printStackTrace();
 }
 }
}

class Computation2 implements Runnable
{
 public static int sum = 0;
 public void run()
 {
 // check if newBarrier is broken or not
 System.out.println("Is the barrier broken? - " +
Tester.newBarrier.isBroken());
 sum = 10 + 20;
 try
 {
 Tester.newBarrier.await(3000,
TimeUnit.MILLISECONDS);

 // number of parties waiting at the barrier
 System.out.println("Number of parties waiting at the
barrier " +
 "at this point = " +
Tester.newBarrier.getNumberWaiting());
 }
 }
}
```

```

 }
 catch (InterruptedException | BrokenBarrierException e)
 {
 e.printStackTrace();
 }
 catch (TimeoutException e)
 {
 e.printStackTrace();
 }
}
}

public class Tester implements Runnable
{
 public static CyclicBarrier newBarrier = new CyclicBarrier(3);

 public static void main(String[] args)
 {
 // parent thread
 Tester test = new Tester();

 Thread t1 = new Thread(test);
 t1.start();
 }

 public void run()
 {
 System.out.println("Number of parties required to trip
the barrier = "+
 newBarrier.getParties());
 System.out.println("Sum of product and sum = " +
(Computation1.product +
 Computation2.sum));

 // objects on which the child thread has to run
 Computation1 comp1 = new Computation1();
 Computation2 comp2 = new Computation2();

 // creation of child thread
 Thread t1 = new Thread(comp1);
 Thread t2 = new Thread(comp2);

 // moving child thread to runnable state
 t1.start();
 t2.start();

 try

```

```

 {
 Tester.newBarrier.await();
 }
 catch (InterruptedException | BrokenBarrierException e)
 {
 e.printStackTrace();
 }

 // barrier breaks as the number of thread waiting for
 the barrier
 // at this point = 3
 System.out.println("Sum of product and sum = " +
(Computation1.product +
 Computation2.sum));

 // Resetting the newBarrier
 newBarrier.reset();
 System.out.println("Barrier reset successful");
}
}
}

```

### **Output:**

```

<Number of parties required to trip the barrier = 3
Sum of product and sum = 0
Is the barrier broken? - false
Number of parties waiting at the barrier at this point = 0
Sum of product and sum = 36
Barrier reset successful

```

**Explanation:** The value of  $(\text{sum} + \text{product}) = 0$  is printed on the console because the child thread hasn't yet ran to set the values of sum and product variable. Following this,  $(\text{sum} + \text{product}) = 36$  is printed on the console because the child threads ran setting the values of sum and product. Furthermore, the number of waiting thread on the barrier reached 3, due to which the barrier then allowed all thread to pass and finally 36 was printed. The value of "Number of parties waiting at the barrier at this point" = 0 because all the three threads had already called await() method and hence, the barrier is no longer active. In the end, newBarrier is reset and can be used again.

### **BrokenBarrierException**

A barrier breaks when any of the waiting thread leaves the barrier. This happens when one or more waiting thread is interrupted or when the waiting time is completed because the thread called the await() methods with a timeout as follows:

```

newBarrier.await(1000, TimeUnit.MILLISECONDS);
// thread calling this await()

```

```
// methods waits for only 1000 milliseconds.
```

When the barrier breaks due to one of more participating threads, the await() methods of all the other threads throws a BrokenThreadException. Whereas, the threads that are already waiting in the barriers have their await() call terminated.

### Difference between a CyclicBarrier and a CountDownLatch

- A CountDownLatch can be used only once in a program(until it's count reaches 0).
- A CyclicBarrier can be used again and again once all the threads in a barriers is released.

## 7. Important point of CyclicBarrier in Java

1. CyclicBarrier can perform a completion task once all thread reaches to the barrier, This can be provided while creating CyclicBarrier.
2. If CyclicBarrier is initialized with 3 parties means 3 thread needs to call await method to break the barrier.
3. The thread will block on await() until all parties reach to the barrier, another thread interrupt or await timed out.
4. If another thread interrupts the thread which is waiting on barrier it will throw BrokenBarrierException as shown below:

```
java.util.concurrent.BrokenBarrierException
 at
java.util.concurrent.CyclicBarrier.dowait (CyclicBarrier.java:1
72)
 at
java.util.concurrent.CyclicBarrier.await (CyclicBarrier.java:32
7)
```

5. CyclicBarrier.reset() put Barrier on its initial state, other thread which is waiting or not yet reached barrier will terminate with java.util.concurrent.BrokenBarrierException.

That's all on What is CyclicBarrier in Java When to use CyclicBarrier in Java and a Simple Example of How to use CyclicBarrier in Java . We have also seen the

difference between `CountDownLatch` and `CyclicBarrier` in Java and got some idea where we can use `CyclicBarrier` in Java Concurrent code.

## 8. What is executor framework in java?

**Answer.** This is very important question to start your interview with. [Executor and ExecutorService](#) are used for following purposes >

- creating thread in java,
- starting threads in java,
- managing whole [life cycle of Threads](#) in java.

Executor creates [pool of threads](#) and manages life cycle of all threads in it.

In Executor framework, **Executor** interface and **ExecutorService** class are most prominently used in java.

[Executor](#) interface defines very important `execute()` method which executes command in java.

[ExecutorService](#) interface extends **Executor** interface.

An Executor interface provides following type of methods >

- methods for managing termination and
- methods that can produce a Future for tracking progress of tasks in java.

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

## 9. What are differences between execute() and submit() method of executor framework in java?

**Answer.** This is basic thread concurrency interview question, beforehand you must know about [Executor Service Framework](#).

| <b>execute()</b> method                                                           | <b>submit()</b> method                                                                  |
|-----------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------|
| <b>execute()</b> method is defined in <a href="#">Executor</a> interface in java. | <b>submit()</b> method is defined in <a href="#">ExecutorService</a> interface in java. |

|                                                                                             |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>It can be used for executing <b>runnable task in java</b>.</p>                           | <p>It can be used for executing <b>runnable task</b> or <b>callable task</b>, submitted callable returns future and Future's get method will return the task's result in java.</p>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| <p><b>Signature of execute method is &gt;</b></p> <p><b>void execute(Runnable task)</b></p> | <p>submit method has 3 forms &gt;</p> <p><b>&lt;T&gt; Future&lt;T&gt; submit(Callable&lt;T&gt; task)</b></p> <p>Submits a callable <b>task</b> for execution.<br/>Method <b>returns</b> a Future which represents pending results of the task.<br/>Once task is completed Future's get method will return the task's result.</p> <p><b>&lt;T&gt; Future&lt;T&gt; submit(Runnable task, T result)</b></p> <p>Submits a Runnable <b>task</b> for execution.<br/>Method <b>returns</b> a Future which represents that task. Once task is completed Future's get method will return <b>result</b>.</p> <p><b>Future&lt;?&gt; submit(Runnable task)</b></p> <p>Submits a Runnable <b>task</b> for execution.<br/>Method <b>returns</b> a Future which represents that task. Once task is completed Future's get method will return null.</p> |

## 10. What is Semaphore in java 7?

**Answer.** This is very important thread concurrency interview question for freshers and experienced. A [semaphore](#) controls access to a shared resource by using permits in java.

- **If permits are greater than zero**, then semaphore **allow access to shared resource**.
- **If permits are zero or less than zero**, then semaphore **does not allow access to shared resource in java**.

These permits are sort of counters, which allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore in java.

### Semaphore has 2 constructors >

- **Semaphore(int permits)**

**permits** is the **initial number of permits available**.

This value can be negative, in which case releases must occur before any acquires will be granted, **permits** is number of threads that can access shared resource at a time.

If **permits** is 1, then only one threads that can access shared resource at a time in java.

- **Semaphore(int permits, boolean fair)**

**permits** is the initial number of permits available.

This value can be negative, in which case releases must occur before any acquires will be granted.

By setting **fair** to **true**, we ensure that **waiting threads are granted a permit in the order in which they requested access**.

### Semaphore's acquire( ) method has 2 forms :

- **void acquire( ) throws InterruptedException**

Acquires a permit if one is available and **reduces the number of available permits by 1**.

If no permit is available then the current thread becomes dormant until

>some other thread calls **release()** method on this semaphore or,

>some other thread interrupts the current thread.

- **void acquire(int permits) throws InterruptedException**

Acquires **permits** number of permits if available and **reduces the number of available permits by permits**.

If **permits** number of permits are not available then the current thread becomes dormant until one of the following things happens -

>some other thread calls release() method on this semaphore and available permits become equal to **permits** or,

>some other thread interrupts the current thread.

### *Semaphore's release( ) method has 2 forms in java :*

- **void release( )**

Releases a permit and **increases the number of available permits by 1**.

For releasing lock by calling release() method it's not mandatory that thread must have acquired permit by calling acquire() method in java.

- **void release(int permits)**

Releases **permits** number of permits and **increases the number of available permits by permits**.

For releasing lock by calling release(int *permits*) method it's not mandatory that thread must have acquired permit by calling acquire()/acquire(int *permit*) method in java.

## 11. How can you implement Producer Consumer pattern using Semaphore in java?

**Answer.** This is tricky thread concurrency interview question for even experienced guys.

**Semaphore** on producer is created with permit =1. So, that producer can get the permit to produce.

**Semaphore** on consumer is created with permit =0. So, that consumer could wait for permit to consume. [because initially producer hasn't produced any product]

Producer gets permit by calling **semaphoreProducer.acquire()** and starts producing, after producing it calls **semaphoreConsumer.release()**. So, that consumer could get the permit to consume.

```
semaphoreProducer.acquire();
```

```
System.out.println("Produced : "+i);
semaphoreConsumer.release();
```

Consumer gets permit by calling `semaphoreConsumer.acquire()` and starts consuming, after consuming it calls `semaphoreProducer.release()`. So, that producer could get the permit to produce.

```
semaphoreConsumer.acquire();
System.out.println("Consumed : "+i);
semaphoreProducer.release();
```

```
import java.util.concurrent.Semaphore;

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
public class ConsumerProducer{

 public static void main(String[] args) {
 Semaphore semaphoreProducer=new Semaphore(1);
 Semaphore semaphoreConsumer=new Semaphore(0);
 System.out.println("semaphoreProducer permit=1 | semaphoreConsumer permit=0");

 Producer producer=new Producer(semaphoreProducer,semaphoreConsumer);
 Consumer consumer=new Consumer(semaphoreConsumer,semaphoreProducer);

 Thread producerThread = new Thread(producer, "ProducerThread");
 Thread consumerThread = new Thread(consumer, "ConsumerThread");
 }
}
```

```
producerThread.start();

consumerThread.start();

}

}

/**

 * Producer Class.

 */

class Producer implements Runnable{

Semaphore semaphoreProducer;

Semaphore semaphoreConsumer;

public Producer(Semaphore semaphoreProducer,Semaphore semaphoreConsumer) {

 this.semaphoreProducer=semaphoreProducer;

 this.semaphoreConsumer=semaphoreConsumer;

}

public void run() {

 for(int i=1;i<=5;i++){

 try {

 semaphoreProducer.acquire();

 System.out.println("Produced : "+i);

 semaphoreConsumer.release();

 } catch (InterruptedException e) {

 e.printStackTrace();

 }

 }

}

}
```

```
/*
 * Consumer Class.
 */

class Consumer implements Runnable{
 Semaphore semaphoreConsumer;
 Semaphore semaphoreProducer;

 public Consumer(Semaphore semaphoreConsumer,Semaphore semaphoreProducer) {
 this.semaphoreConsumer=semaphoreConsumer;
 this.semaphoreProducer=semaphoreProducer;
 }

 public void run() {

 for(int i=1;i<=5;i++){
 try {
 semaphoreConsumer.acquire();
 System.out.println("Consumed : "+i);
 semaphoreProducer.release();
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 }
}

/*OUTPUT
semaphoreProducer permit=1 | semaphoreConsumer permit=0
Produced : 1
Consumed : 1
Produced : 2
```

```

Consumed : 2
Produced : 3
Consumed : 3
Produced : 4
Consumed : 4
Produced : 5
Consumed : 5
*/

```

Let's discuss output in detail, to get better understanding of how we have used Semaphore for implementing Producer Consumer pattern >

**Note :** (I have mentioned output in **green** text and it's explanation is given in line immediately followed by it)

`semaphoreProducer permit=1 | semaphoreConsumer permit=0`

semaphoreProducer created with permit=1. So, that producer can get the permit to produce |  
 semaphoreConsumer created with permit=0. So, that consumer could wait for permit to consume.

`semaphoreProducer.acquire()` is called, Producer has got the permit and it can produce [Now, `semaphoreProducer permit=0`]

`Produced : 1` [as producer has got permit, it is producing]

`semaphoreConsumer.release()` is called, Permit has been released on `semaphoreConsumer` means consumer can consume [Now, `semaphoreConsumer permit=1`]

`semaphoreConsumer.acquire()` is called, Consumere has got the permit and it can consume [Now, `semaphoreConsumer permit=0`]

`Consumed : 1` [as consumer has got permit, it is consuming]

`semaphoreProducer.release()` is called, Permit has been released on `semaphoreProducer` means producer can produce [Now, `semaphoreProducer permit=1`]

```

Produced : 2
Consumed : 2
Produced : 3
Consumed : 3
Produced : 4
Consumed : 4
Produced : 5
Consumed : 5

```

## 12. How can you implement your own Semaphore?

**Answer.** Experienced developers must be able to answer this thread concurrency interview question.

1) Custom Semaphore in java >

In previous tutorial we read how to use [Semaphore in java](#). In this post we will be implementing **custom Semaphore**. This post intends you give you basic functionality of Semaphore using your own java code

A custom **semaphore** controls access to a shared resource by using permits.

- If permits are greater than zero, then semaphore **allow access to shared resource**.
- If permits are zero or less than zero, then semaphore **does not allow access to shared resource**.

These permits are sort of counters, which allow access to the shared resource. Thus, to access the resource, a thread must be granted a permit from the semaphore.

1.1) Custom Semaphore's **constructors** in java >

- **SemaphoreCustom (int permits)**  
**permits** is the initial number of permits available.

This value can be negative, in which case releases must occur before any acquires will be granted, **permits** is number of threads that can access shared resource at a time.

If **permits** is 1, then only one threads that can access shared resource at a time.

**CODE >**

```
public SemaphoreCustom(int permits) {
 this.permits=permits;
}
```

1.2) Custom Semaphore's **acquire()** method :

- **void acquire( ) throws InterruptedException**

Acquires a permit if one is available and **decrements the number of available permits by 1**.

If no permit is available then the current thread waits until one of the following things happen >

- >some other thread calls release() method on this semaphore or,
- >some other thread interrupts the current thread.

**CODE >**

```
public synchronized void acquire() throws InterruptedException {
 //Acquires a permit, if permits is greater than 0 decrements
 //the number of available permits by 1.
 if(permits > 0){
 permits--;
 }

 //permit is not available wait, when thread
 //is notified it decrements the permits by 1
}
```

```

 else{
 this.wait();
 permits--;
 }
}

```

### 1.3) Custom Semaphore's **release()** method :

- **void release( )**

Releases a permit and **increases the number of available permits by 1.**

For releasing lock by calling release() method it's not mandatory that thread must have acquired permit by calling acquire() method.

#### **CODE >**

```

public synchronized void release() {
 //increases the number of available permits by 1.
 permits++;

 //If permits are greater than 0, notify waiting threads.
 if(permits > 0)
 this.notify();
}

```

### 2) Custom Semaphore's code in java >

```

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
/**
 * @author AnkitMittal
 * Copyright (c), AnkitMittal .

```

```

* All Contents are copyrighted and must not be reproduced in any form.

* A semaphore controls access to a shared resource by using permits.

 - If permits are greater than zero, then semaphore
 allow access to shared resource.

 - If permits are zero or less than zero, then semaphore
 does not allow access to shared resource.

*/
class SemaphoreCustom{

 private int permits;

 /** permits is the initial number of permits available.

 This value can be negative, in which case releases must occur
 before any acquires will be granted, permits is number of threads
 that can access shared resource at a time.

 If permits is 1, then only one threads that can access shared
 resource at a time.

 */
 public SemaphoreCustom(int permits) {
 this.permits=permits;
 }

 /**Acquires a permit if one is available and decrements the
 number of available permits by 1.

 If no permit is available then the current thread waits
 until one of the following things happen >
 >some other thread calls release() method on this semaphore or,
 >some other thread interrupts the current thread.

 */
 public synchronized void acquire() throws InterruptedException {
 //Acquires a permit, if permits is greater than 0 decrements

```

```

//the number of available permits by 1.

if(permits > 0){

 permits--;
}

//permit is not available wait, when thread
//is notified it decrements the permits by 1

else{

 this.wait();

 permits--;
}

}

/** Releases a permit and increases the number of available permits by 1.

For releasing lock by calling release() method it's not mandatory
that thread must have acquired permit by calling acquire() method.

*/
public synchronized void release() {

 //increases the number of available permits by 1.

 permits++;

 //If permits are greater than 0, notify waiting threads.

 if(permits > 0)

 this.notify();
}
}

```

**3) Program to demonstrate usage of Custom Semaphore in java >**

```

/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */

/*
* @author AnkitMittal

```

```

* Copyright (c), AnkitMittal .
* All Contents are copyrighted and must not be reproduced in any form.
* A semaphore controls access to a shared resource by using permits.
 - If permits are greater than zero, then semaphore
 allow access to shared resource.
 - If permits are zero or less than zero, then semaphore
 does not allow access to shared resource.
*/
class SemaphoreCustom{

 private int permits;

 /** permits is the initial number of permits available.
 This value can be negative, in which case releases must occur
 before any acquires will be granted, permits is number of threads
 that can access shared resource at a time.
 If permits is 1, then only one threads that can access shared
 resource at a time.
 */
 public SemaphoreCustom(int permits) {
 this.permits=permits;
 }
 /**Acquires a permit if one is available and decrements the
 number of available permits by 1.
 If no permit is available then the current thread waits
 until one of the following things happen >
 >some other thread calls release() method on this semaphore or,
 >some other thread interrupts the current thread.
 */
 public synchronized void acquire() throws InterruptedException {

```

```
//Acquires a permit, if permits is greater than 0 decrements
//the number of available permits by 1.

if(permits > 0){

 permits--;
}

//permit is not available wait, when thread
//is notified it decrements the permits by 1
else{

 this.wait();
 permits--;
}

}

/** Releases a permit and increases the number of available permits by 1.
 For releasing lock by calling release() method it's not mandatory
 that thread must have acquired permit by calling acquire() method.

*/
public synchronized void release() {

 //increases the number of available permits by 1.
 permits++;

 //If permits are greater than 0, notify waiting threads.
 if(permits > 0)
 this.notify();
}

}

/**
 * Main class, for testing SemaphoreCustom
 */
public class SemaphoreCustomTest {
```

```

static int SharedValue=0;

public static void main(String[] args) {
 SemaphoreCustom semaphore=new SemaphoreCustom(1);
 System.out.println("Semaphore with 1 permit has been created");

 IncrementThread incrementThread=new IncrementThread(semaphore);
 new Thread(incrementThread, "incrementThread").start();

 DecrementThread decrementThread=new DecrementThread(semaphore);
 new Thread(decrementThread, "decrementThread").start();
}

}

class IncrementThread implements Runnable{
 SemaphoreCustom semaphoreCustom;

 public IncrementThread(SemaphoreCustom semaphoreCustom) {
 this.semaphoreCustom=semaphoreCustom;
 }

 public void run(){
 System.out.println(Thread.currentThread().getName()+
 " is waiting for permit");
 try {
 semaphoreCustom.acquire();
 System.out.println(Thread.currentThread().getName()+
 " has got permit");
 }
 }
}

```

```
for(int i=0;i<5;i++){
 Thread.sleep(1000);
 System.out.println(Thread.currentThread().getName()+
 " > "+SemaphoreCustomTest.SharedValue++);
}

} catch (InterruptedException e) {
 e.printStackTrace();
}

System.out.println(Thread.currentThread().getName()+
 " has released permit");
semaphoreCustom.release();

}

}

class DecrementThread implements Runnable{
 SemaphoreCustom semaphoreCustom;
 public DecrementThread(SemaphoreCustom semaphoreCustom){
 this.semaphoreCustom=semaphoreCustom;
 }

 public void run(){
 System.out.println(Thread.currentThread().getName()+
 " is waiting for permit");
 }
}
```

```
try {
 semaphoreCustom.acquire();
 System.out.println(Thread.currentThread().getName() +
 " has got permit");

 for(int i=0;i<5;i++){
 Thread.sleep(1000);
 System.out.println(Thread.currentThread().getName() +
 " >" +SemaphoreCustomTest.SharedValue--);
 }

} catch (InterruptedException e) {
 e.printStackTrace();
}

System.out.println(Thread.currentThread().getName() +
 " has released permit");
semaphoreCustom.release();
}
}
/*OUTPUT
Semaphore with 1 permit has been created
incrementThread is waiting for permit
incrementThread has got permit
decrementThread is waiting for permit
incrementThread > 0
```

```

incrementThread > 1
incrementThread > 2
incrementThread > 3
incrementThread > 4
incrementThread has released permit
decrementThread has got permit
decrementThread >5
decrementThread >4
decrementThread >3
decrementThread >2
decrementThread >1
decrementThread has released permit
*/

```

### 13. What is significance of atomic classes in java 7?

**Answer.** Another important and basic thread concurrency interview question for freshers. Java provides some classes in [java.util.concurrent.atomic](#) which offers an alternative to the other [synchronization](#) in java.

*Classes found in [java.util.concurrent.atomic](#) are >*

- **AtomicInteger,**
- **AtomicLong,** and
- **AtomicBoolean.**

*Methods provided by these classes >*

- **get( ),**
- **set( ),**
- **getAndSet( ),**

- **compareAndSet( ), and**
- **decrementAndGet( ).**

In [multithreading](#) environment we can use these classes without any explicit synchronization, as all these classes are [thread safe](#) in java.

For more information on atomic read [Atomic operations in java](#).

#### 14. What are Future and Callable? How are they related in java?

**Answer.** This is **very very important** thread concurrency interview question. They are widely used in thread concurrency.

[\*Future<V>\*](#) interface provides methods >

- for **returning result** of computation, wait until computation is not completed and
- for **cancelling** the computation in between.

[\*Future Methods\*](#) >

[\*V get\(\)\*](#) method returns the result of computation, method waits for computation to complete.

[\*cancel method\*](#) cancels the task.

[\*Callable<V>\*](#) interface provides method for computing a result and returning that computed result or throws an exception if unable to do so

Any class implementing Callable interface must override [\*call\(\)\*](#) method for computing a result.

Method returns computed result or throws an exception if unable to do so in java.

[\*what type of results Callable's call\(\) method can return in java?\*](#)

The Callable<V> is a generic interface, so its call method can return generic result specified by [\*V\*](#).

[\*How Callable and Future are related?\*](#)

If you submit a Callable object to an Executor returned object is of Future type.

```
Future<Double> futureDouble=executor.submit(new SquareDoubleCallable(2.2));
```

where, `SquareDoubleCallable` is a class which implements Callable.

This Future object can check the status of a Callable call's method and wait until Callable's call() method is not completed.

## 15. Similarity and differences between java.util.concurrent.Callable and java.lang.Runnable in java?

**Answer.** This is basic thread concurrency interview question.

[Similarity between java.util.concurrent.Callable and java.lang.Runnable in java?](#)

Instances of class which implements callable are executed by another thread.

[Difference between java.util.concurrent.Callable and java.lang.Runnable in java?](#)

Class implementing Callable interface must override call() method. call() method returns computed result or throws an exception if unable to do so.

Class implementing Runnable interface must override run() method.

A Runnable does not return a result and can neither throw a checked exception in java.

## 16. What is CountDownLatch in java?

**Answer.** This is very important thread concurrency interview question. Fresher and experienced both be well versed with this. There might be situation where we might like our thread to wait until one or more threads completes certain operation in java.

A `CountDownLatch` is initialized with a given `count`.

**count** specifies the number of events that must occur before latch is released.

Every time a event happens **count** is reduced by 1. Once count reaches 0 latch is released.

### *CountDownLatch's constructor >*

- **CountDownLatch(int count)**

CountDownLatch is initialized with given **count**.

**count** specifies the number of events that must occur before latch is released.

### *CountDownLatch's await() method has 2 forms :*

- **void await( ) throws InterruptedException**

Causes the current thread to wait until one of the following things happens-

- latch **count** has down to reached 0, or
- unless the thread is interrupted.

- **boolean await(long timeout, TimeUnit unit)**

Causes the current thread to wait until one of the following things happens-

- latch **count** has down to reached 0,
- unless the thread is interrupted, or
- specified **timeout** elapses.

### *CountDownLatch's countDown() method in java :*

- **void countDown( )**

Reduces latch **count** by 1.

If **count** reaches 0, all waiting threads are released.

Read more about [CountDownLatch in java](#).

## 17. Where can you use CountDownLatch in real world?

**Answer.** Very interesting question. It will test your real time thread concurrency implementation skills. When you go in amusement park, you must have seen on certain rides there is mandate that at least 3 people (**3 is count**) should be there to take a ride. So, ride keeper (**ride keeper is main thread**) waits for 3 persons (**ride keeper has called await()**).

Every time a person comes count is reduced by 1 (**let's say every person is calling countDown() method**). Ultimately when 3 persons reach count becomes 0 & wait for ride keeper comes to end.

## 18. How can you implement your own CountDownLatch in java?

**Answer.**

There might be situation where we might like our thread to wait until one or more threads completes certain operation.

A CountDownLatch is initialized with a given **count**.

**count** specifies the number of events that must occur before latch is released.

Every time a event happens **count** is reduced by 1. Once count reaches 0 latch is released.

1.1) Custom CountDownLatch's **constructor** in java >

- **CountDownLatch(int count)**

CountDownLatch is initialized with given **count**.

**count** specifies the number of events that must occur before latch is released.

**CODE >**

```
public CountDownLatchCustom(int count) {
 this.count=count;
}
```

1.2) Custom CountDownLatch's **await()** method in java:

- **void await( ) throws InterruptedException**

Causes the current thread to wait until one of the following things happens-

- latch **count** has down to reached 0, or

- unless the thread is interrupted.

**CODE >**

```
public synchronized void await() throws InterruptedException {
 //If count is greater than 0, thread waits.
 if(count>0)
 this.wait();
}
```

1.3) Custom CountDownLatch's **countDown()** method in java:

- **void countDown( )**

Reduces latch **count** by 1.

If **count** reaches 0, all waiting threads are released.

**CODE >**

```
public synchronized void countDown() {
 //decrement the count by 1.
 count--;

 //If count is equal to 0, notify all waiting threads.
 if(count == 0)
 this.notify();
}
```

2) Custom CountDownLatch's code in java >

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
```

```
/**
 * @author AnkitMittal
 * Copyright (c), AnkitMittal .
 * All Contents are copyrighted and must not be reproduced in any form.
 CountDownLatchCustom wait until one or more threads completes certain operation.
 A CountDownLatch is initialized with a given count .
 count specifies the number of events that must occur before
 latch is released.
 Every time a event happens count is reduced by 1. Once count
 reaches 0 latch is released.
 */

class CountDownLatchCustom{
 private int count;
 /**
 * CountDownLatch is initialized with given count.
 * count specifies the number of events that must occur
 * before latch is released.
 */
 public CountDownLatchCustom(int count) {
 this.count=count;
 }
 /**
 * Causes the current thread to wait until one of the following things happens-
 * - latch count has down to reached 0, or
 * - unless the thread is interrupted.
 */
 public synchronized void await() throws InterruptedException {
 //If count is greater than 0, thread waits.
 if(count>0)
 this.wait();
 }
}
```

```

}

/**
 * Reduces latch count by 1.
 * If count reaches 0, all waiting threads are released.
 */

public synchronized void countDown() {
 //decrement the count by 1.
 count--;

 //If count is equal to 0, notify all waiting threads.
 if(count == 0)
 this.notify();
}

}

```

2) Program to demonstrate usage of Custom CountDownLatch in java >

```

/**
 * @author AnkitMittal
 * Copyright (c), AnkitMittal .
 * All Contents are copyrighted and must not be reproduced in any form.
 CountDownLatchCustom wait until one or more threads completes certain operation.
 A CountDownLatch is initialized with a given count .
 count specifies the number of events that must occur before
 latch is released.
 Every time a event happens count is reduced by 1. Once count
 reaches 0 latch is released.
*/
class CountDownLatchCustom{

```

```
private int count;

/**
 * CountDownLatch is initialized with given count.
 * count specifies the number of events that must occur
 * before latch is released.
 */

public CountDownLatchCustom(int count) {
 this.count=count;
}

/**
 * Causes the current thread to wait until one of the following things happens-
 * - latch count has down to reached 0, or
 * - unless the thread is interrupted.
 */

public synchronized void await() throws InterruptedException {
 //If count is greater than 0, thread waits.
 if(count>0)
 this.wait();
}

/**
 * Reduces latch count by 1.
 * If count reaches 0, all waiting threads are released.
 */

public synchronized void countDown() {
 //decrement the count by 1.
 count--;

 //If count is equal to 0, notify all waiting threads.
 if(count == 0)
 this.notify();
}
```

```
 }

}

/**
* Main class
*/
public class CountDownLatchCustomTest {

 public static void main(String[] args) {
 CountDownLatchCustom countDownLatchCustom=new
CountDownLatchCustom(3);
 System.out.println("CountDownLatch has been created with count=3");

 new Thread(new MyRunnable(countDownLatchCustom), "Thread-1").start();

 try {
 countDownLatchCustom.await();
 } catch (InterruptedException e) {
 e.printStackTrace();
 }

 System.out.println("count has reached zero, "+
Thread.currentThread().getName()+" thread has ended");
 }
}

class MyRunnable implements Runnable{

 CountDownLatchCustom countDownLatchCustom;
```

```
MyRunnable(CountDownLatchCustom countDownLatchCustom){
 this.countDownLatchCustom=countDownLatchCustom;
}

public void run(){

 for(int i=2;i>=0;i--){

 countDownLatchCustom.countDown();
 System.out.println(Thread.currentThread().getName() +
 " has reduced latch count to : "+ i);

 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
}
}
/*OUTPUT
CountDownLatch has been created with count=3
Thread-1 has reduced latch count to : 2
Thread-1 has reduced latch count to : 1
Thread-1 has reduced latch count to : 0
count has reached zero, main thread has ended
*/
```

2.1) Let's discuss output in detail, to get better understanding of **Custom CountDownLatch** usage in program >

**Note** : I have mentioned output in **green** text.

CountDownLatch has been created with count=3  
 Initially, **custom CountDownLatch** is created with count=3

main thread called **countDownLatchCustom.await()** and it is waiting for count to become 0.

Thread-1 called **countDownLatchCustom.countDown()** method. [Now, count=2]

Thread-1 has reduced latch count to : 2

Thread-1 called **countDownLatchCustom.countDown()** method. [Now, count=1]

Thread-1 has reduced latch count to : 1

Thread-1 called **countDownLatchCustom.countDown()** method. [Now, count=0]

Thread-1 has reduced latch count to : 0

count has reached zero, main thread has ended

As, count has reached zero, main thread has ended.

2.3) Occasionally, because of threads unpredictable behaviour output may be bit awkward in java >

```
/*OUTPUT
CountDownLatch has been created with count=3
Thread-1 has reduced latch count to : 2
Thread-1 has reduced latch count to : 1
count has reached zero, main thread has ended
Thread-1 has reduced latch count to : 0
*/
```

This may happen because as soon as count reaches 0 waiting threads are released. Here, as soon as Thread-1 called countDown() method third time main thread was released and its sysout statement was executed before Thread-1's sysout statement.

## 19. What is CyclicBarrier in java?

**Answer.** This is very important thread concurrency interview question. Fresher and experienced both be well versed with this. There might be situation where we might have to trigger event only when one or more threads completes certain operation in java.

**2 or more threads wait for each other to reach a common barrier point.** When all **threads** have **reached** common **barrier point** (i.e. when all threads have called await() method) >

- **All waiting threads are released**, and
- **Event can be triggered** as well.

### CyclicBarrier's constructor in java >

- **CyclicBarrier(int parties)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released.

- **CyclicBarrier(int parties, Runnable barrierAction)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction** (event) is triggered.

### CyclicBarrier's await() method has 2 forms :

- **int await() throws InterruptedException, BrokenBarrierException**

If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -

- The last thread to call arrive(i.e. call await() method), or
- Some other thread interrupts the current thread, or
- Some other thread interrupts one of the other waiting threads, or
- Some other thread times out while waiting for barrier, or

- Some other thread invokes reset() method on this cyclicBarrier.
  
- `int await(long timeout, TimeUnit unit)` throws `InterruptedException`, `BrokenBarrierException`, `TimeoutException`

If the current thread is not the last to arrive(i.e. call await() method) then it waits until one of the following things happens -

  - The last thread to call arrive(i.e. call await() method), or
  - The specified `timeout` elapses, or
  - Some other thread interrupts the current thread, or
  - Some other thread interrupts one of the other waiting threads, or
  - Some other thread times out while waiting for barrier, or
  - Some other thread invokes reset() method on this cyclicBarrier.

Read more about [CyclicBarrier in java](#).

## 20. Why is CyclicBarrier cyclic in java?

**Answer.** This is very interesting thread concurrency interview question for developers. The barrier is called *cyclic* because [CyclicBarrier](#) can be reused after -

- All the waiting threads are released in java and
- event has been triggered in java.

## 21. Where could we use CyclicBarrier in real world?

**Answer.** Another very interesting question. It will test your real time thread concurrency implementation skills. Let's say 10 friends (**friends are threads**) have planned for picnic on place A (Here **place A is common barrier point**). And they all decided to play certain game (**game is event**) only on everyones arrival at place A. So, all 10 friends must wait for each other to reach place A before launching event.

Now, when all **threads** have **reached common barrier point** (i.e. all friends have reached place A) >

- **All waiting threads are released** (All friends can play game), and
- **Event can be triggered** (they will start playing game).

## 22. How can you implement your own CyclicBarrier in java?

**Answer.** This is another important and complex interview for developers. Please see [Implementation of custom/own CyclicBarrier in java](#).

## Thread concurrency interview Question 16. Similarity and Difference between CyclicBarrier and CountDownLatch in Java?

**Answer.** This is **very very important** thread concurrency interview question. Fresher and experienced both must be well versed with this.

1. **CyclicBarrier** and **CountDownLatch** are similar because they wait for specified number of threads to reach certain point and make count/parties equal to 0. But, for completing wait in CountDownLatch specified number of threads must call **countDown()** method in java.  
for completing wait in CyclicBarrier specified number of threads must call **await()** method.
2. Let's see their constructor's >

| <b>CountDownLatch(int count)</b>                                                                                                                           | <b>CyclicBarrier(int parties)</b>                                                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <p>CountDownLatch is initialized with given <b>count</b>.</p> <p><b>count</b> specifies the number of events that must occur before latch is released.</p> | <p>New CyclicBarrier is created where <b>parties</b> number of threads wait for each other to reach common barrier point, when all threads have reached common barrier point, <b>parties</b> number of waiting threads are released.</p> |

3. **CyclicBarrier** can be awaited repeatedly, but **CountDownLatch** can't be awaited repeatedly. i.e. once count has become 0 cyclicBarrier can be used again but CountDownLatch cannot be used again in java.
4. **CyclicBarrier** can be used to trigger event, but **CountDownLatch** can't be used to launch event. i.e. once count has become 0 cyclicBarrier can trigger event but CountDownLatch can't in java.

### How can cyclicBarrier launch event?

CyclicBarrier provides constructor for triggering event.

#### **CyclicBarrier(int parties, Runnable barrierAction)**

New CyclicBarrier is created where **parties** number of thread wait for each other to reach common barrier point, when all threads have reached common barrier point, **parties** number of waiting threads are released and **barrierAction (event) is triggered**.

## Thread concurrency interview Question 17. What is Phaser in java? Is Phaser similar to CyclicBarrier?

**Answer.** This is another very important thread concurrency interview question. [Phaser](#) is somewhat **similar** in functionality of [CyclicBarrier](#) and [CountDownLatch](#) but it provides more flexibility than both of them.

Phaser provides us flexibility of registering and deRegistering parties at any time.

**For registering parties**, we may use any of the following -

- constructors, or
- int register(), or
- bulkRegister().

**For deRegistering parties**, we may use any of the following -

- arriveAndDeregister()

we have methods like **getPhase()** which returns the current phase number. And

**isTerminated()** method returns **true** if phaser has been **terminated**.

Read more about [Phaser in java](#)

## Thread concurrency interview Question 18. Differences and similarity between Phaser and CyclicBarrier in java?

**Answer.** Another interesting thread concurrency interview question. Like a [CyclicBarrier](#), a [Phaser](#) can be **awaited repeatedly**.

But, in CyclicBarrier we used to register parties in constructor but Phaser provides us flexibility of registering and deRegistering parties at any time in java.

## Thread concurrency interview Question 19.Difference between arrive() and arriveAndAwaitAdvance() method of Phaser in java?

**Answer.** This is thread concurrency interview question for experienced developers. **arrive()** method of [Phaser](#) **does not cause current thread to wait for other registered threads to complete current phase**. That means current thread can immediately start next phase without waiting for any other registered thread to complete current phase.

But, **arriveAndAwaitAdvance() method causes current thread to wait for other registered threads to complete current phase**. That means current thread can proceed to next phase only when all other threads have completed current phase (i.e. by calling **arriveAndAwaitAdvance() method**).

## Thread concurrency interview Question 20. When is phaser terminated in java?

**Answer.** This is another thread concurrency interview question for experienced developers. When calling **arriveAndDeregister()** method of [Phaser](#) has caused the number of registered parties to become 0. Termination can also be triggered when an **onAdvance()** method returns **true**.

## Question 21. How can you control number of phase you want to execute in Phaser in java?

**Answer.** Yet another thread concurrency interview question for experienced developers. We can override the `onAdvance( )` method of `Phaser` to control number of phases which we want to execute.

Signature of `onAdavance` method is `boolean onAdvance(int phase, int registeredParties)`.

Where, `phase` is the current phase number when we enter `onAdvance()` method i.e. before advancing to next phase.

`registeredParties` is the current number of registered parties

**Every Time before advancing to next phase overridden `onAdvance()` method is called** and returns either true or false.

If method returns `true` than `phaser is terminated` ,or

If method returns `false` then `phaser continues` and can `advance to next phase`.

[Program to demonstrate usage of how we can override Phaser's `onAdvance` method to control number of phase we want to execute](#)

## Thread concurrency interview Question 22. Where could we use Phaser in real world?

**Answer.** Another interesting thread concurrency interview question. Software process management is done in phases.

- First phase could be **requirement gathering**,
- second could be **software development** and
- third could be **testing**.

Second phase will not start until first is not completed, like wise third phase will not start until second is not completed.

## Thread concurrency interview Question 23. What is maximum number of parties that could be registered with phaser at a time in java ?

**Answer.** Complex and challenging interview question even for experienced. Maximum number of parties that could be registered with phaser at a time is **65535**, if we try to register more parties **IllegalStateException** will be thrown in java.

## Thread concurrency interview Question 24. What is exchanger in Java?

**Answer.** This is very important thread concurrency interview question for freshers and experienced. [Exchanger](#) enables two threads to exchange their data between each other. Exchanger can be handy in solving Producer Consumer pattern where Producer and consumer threads can exchange their data.

- **exchange(V x)**  
exchange() method enables two threads to exchange their data between each other.  
**If current thread is first one to call exchange()** method then it will wait until one of following things happen >
  - Some other thread calls exchange() method, or
  - Some other thread interrupts the current thread, or

**If some other thread has already called exchanger() method then it resumes its execution and following things happen -**

- waiting thread is resumed and receives data from current thread.
  - current thread receives data from that waiting thread and it returns immediately.
- **V exchange(V x, long timeout, TimeUnit unit)**  
exchanger() method enables two threads to exchange their data between each other.

**If current thread is first one to call exchange() method then it will wait until one of following things happen >**

- Some other thread calls exchange() method, or
- Some other thread interrupts the current thread, or
- The specified **timeout** elapses.

**If some other thread has already called exchanger() method then it resumes its execution and following things happen -**

- waiting thread is resumed and receives data from current thread.
- current thread receives data from that waiting thread and it returns immediately.

Read more about [Exchanger in java](#).

## Thread concurrency interview Question 25. How can you implement Producer Consumer pattern using Exchanger in java?

**Answer.** Very interesting thread concurrency interview question for experienced developers. Exchanger is created, which will enable Producer and consumer threads to exchange their data.

Producer thread produces and called exchanger() method, now it will wait for consumer thread to call exchange() method.

Consumer thread calls exchanger() method and following things will happens >

- current thread(consumerThread) will receive data from that waiting thread(producerThread) and it returns immediately.
- waiting thread (producerThread) will resume and receive data from current thread (consumerThread).

[Read program to implement Producer Consumer pattern using Exchanger](#)

## Question 26. How can you solve consumer producer pattern by using BlockingQueue in java?

**Answer.** It is **very very important** thread concurrency interview question for all developers.. Now it's time to gear up to face question which is most probably going to be followed up by previous question i.e. after how to solve consumer producer problem using [wait\(\) and notify\(\) method](#). Generally you might wonder why interviewer's are so much interested in asking about [solving consumer producer problem using BlockingQueue](#), answer is they want to know how strong knowledge you have about java concurrent Api's, this Api use consumer producer pattern in very optimized manner, BlockingQueue is designed is such a manner that it offer us the best performance.

[BlockingQueue is a interface](#) and we will use its [implementation class LinkedBlockingQueue](#).

Key methods for solving consumer producer pattern are >

```
put(i); //used by producer to put/produce in sharedQueue.
take(); //used by consumer to take/consume from sharedQueue.
```

## Question 27. How can you implement your own LinkedBlockingQueue to solve consumer producer pattern in java?

**Answer.** Another challenging, logical and complex thread concurrency interview question, Please read [Producer Consumer pattern using Custom implementation of BlockingQueue interface](#)

## Thread concurrency interview Question 28. What is Lock in java?

**Answer.** Important thread concurrency interview question. The `java.util.concurrent.locks.Lock`s is a interface and its implementations provide more extensive locking operations than can be obtained using synchronized methods and statements.

**A lock helps in controlling access to a shared resource by multiple threads. Only one thread at a time can acquire the lock and access the shared resource in java.**

If a second thread attempts to acquire the lock on shared resource when it is acquired by another thread, the second thread will wait until the lock is released. In this way we can achieve [synchronization](#) and [race conditions](#) can be avoided in java.

Read lock of a ReadWriteLock may allow concurrent access to a shared resource in java.

Read more about [locks](#) and [ReentrantLocks](#) in java

## Thread concurrency interview Question 29. Explain key methods of Lock interface in java?

**Answer.** This is very important thread concurrency interview question for freshers and experienced.

[\*Lock interface key methods in java >\*](#)

### **void lock()**

Acquires the lock if it is not held by another thread. **And sets lock hold count to 1.**

If current thread already holds lock then **lock hold count** is increased by 1.

If the lock is held by another thread then the current thread waits for another thread to release lock.

## *void unLock()*

If the current thread is the holding the lock then the **lock hold count** is decremented by 1. If the **lock hold count** has reached 0, then the lock is released.

If **lock hold count** is still greater than 0 then lock is not released.

If the current thread is not holding the lock then **IllegalMonitorStateException** is thrown.

## *boolean tryLock()*

Acquires the lock if it is not held by another thread and returns true. **And sets lock hold count to 1.**

If current thread already holds lock then method returns true. **And increments lock hold count by 1.**

If lock is held by another thread then method return false.

## *boolean tryLock(long timeout, TimeUnit unit)*

### *throws InterruptedException*

Acquires the lock if it is not held by another thread and returns true. **And sets lock hold count to 1.**

If current thread already holds lock then method returns true. **And increments lock hold count by 1.**

If lock is held by another thread then current thread will wait until one of the following things happen -

- **Another thread releases lock and the lock is acquired by the current thread,**
- or**
- **Some other thread interrupts the current thread, or**
- **The specified timeout elapses .**

If the **lock is acquired** then method **returns true**. **And sets lock hold count to 1.**

If **specified timeout elapses** then method return false.

## Condition newCondition()

Method returns a Condition instance to be used with this Lock instance.

Condition instance are similar to using [Wait\(\), notify\(\) and notifyAll\(\)](#) methods.

- **IllegalMonitorStateException** is thrown if this lock is not held when any of the Condition waiting or signalling methods are called.
- **Lock is released** when the condition waiting methods are called and before they return, the lock is reacquired and the **lock hold count** restored to what it was when the method was called.
- If a thread is interrupted while waiting then **InterruptedException** will be thrown and following things will happen -
  - the wait will be over, and
  - thread's interrupted status will be cleared.
- Waiting threads are signalled in FIFO (first in first out order) order.
- When lock is **fair**, first lock is obtained by longest-waiting thread.

If lock is not **fair**, any waiting thread could get lock, at discretion of implementation.

Read more about [locks in java](#)

**Thread concurrency interview Question 30. Explain usage of newCondition() method of Lock interface in detail in java? And can it be used to implement producer consumer pattern in java?**

6. Answer. It is very complex thread concurrency interview question. Even most of the experienced developers are not aware of this question. Please read [ReentrantLock class provides implementation of Lock's newCondition\(\) method in java - description and solving producer consumer program using this method.](#)

## Thread concurrency interview Question 31. Explain key methods of ReentrantLock class in java?

**Answer.** This is very important thread concurrency interview question for freshers and experienced.

ReentrantLock class provides implementation of all Lock interface methods

*void lock()*

*void unLock()*

*boolean tryLock()*

*boolean tryLock(long timeout, TimeUnit unit)*

Additional methods provided by ReentrantLock class are >

*void lockInterruptibly() throws InterruptedException*

If current thread already holds lock then method returns true. And increments **lock hold count** by 1.

If the lock is held by another thread then the current thread waits until one of the following thing happens -

- The lock is acquired by the current thread, or
- Some other thread **interrupts the current thread**.

As soon as current thread acquires the lock it **sets lock hold count** to 1.

## *int getWaitQueueLength(Condition condition)*

Method returns number of threads that may be waiting to acquire this lock.

Method is used just for monitoring purposes and not for any kind of synchronization purposes.

## *boolean isHeldByCurrentThread()*

Method returns true if lock is held by current thread. Its similar to **Thread.holdsLock()** method.

Read more about [ReEntrantLocks in java](#)

## **Thread concurrency interview Question 32. Write Program to demonstrate usage of ReentrantLock in java?**

**Answer.** Interesting question for developers. [Read program to demonstrate usage of ReentrantLock](#).

## **Thread concurrency interview Question 33. How can you implement your own ReentrantLock in java?**

**Answer.** This is important and complex interview question for developers. [Implementation of custom/own Lock and ReentrantLock in java](#)

## Thread concurrency interview Question 34. What is Fork/Join Framework in java ?

**Answer.** This is **very very important** thread concurrency interview question. Fresher and experienced both must be well versed with this..

*Fork/Join Framework has been added in JDK 7 and is defined in the `java.util.concurrent` package in java.*

*Fork/Join framework enables **parallel programming**. Parallel programming means taking **advantage two or more processors (multicore) in computers**. Parallel programming improves program performance in java.*

*The Fork/Join Framework also **improves program performance** in following ways >*

- Fork/Join framework makes use of multiple processors available in computer. Hence enabling parallel processing, and
- It managing whole [life cycle of Threads](#).

7. [Read more about Fork/Join Framework - Parallel programming in java.](#)

## Thread concurrency interview Question 35. What is Divide-and-conquer in Fork/Join framework in java ?

**Answer.** Basic thread concurrency interview question, which tests the developers in depth knowledge about the fork join framework in java. The **divide-and-conquer** strategy recursively divides a task into smaller subtasks until subtask isn't small enough to be solved independently.

## Thread concurrency interview Question 36. What approach does ForkJoinPool uses for managing tasks in java?

**Answer.** Another basic thread concurrency interview question, which tests the developers in depth knowledge about the fork join framework in java. **ForkJoinPool** uses ***work-stealing approach*** for managing threads. Each thread in ForkJoinPool maintains a queue of tasks. If one thread's queue is empty, it can take task from another thread. This overall improves the program/applications performance in java.

## Thread concurrency interview Question 37. What are ForkJoinPool and ForkJoinTask in java?

Answer.

### ForkJoinPool in java

**ForkJoinPool** implements ExecutorService framework. The execution of **ForkJoinTasks** takes place within a **ForkJoinPool**, which also manages the execution of the tasks.

### *ForkJoinPool constructors >*

- ***ForkJoinPool( )***
  - Creates a pool in java.
  - **level of parallelism = number of processors available in the system**
- ***ForkJoinPool(int parallelism)***
  - The **parallelism** is the **level of parallelism**. Its value must be greater than 0 and must not be more than number of processors in system.
  - **level of parallelism** determines the number of threads that can execute simultaneously. As a number of threads are determined it also determines number of tasks that could be executed **parallelly** in java.

## ForkJoinPool important methods in java >

After you have created an instance of **ForkJoinPool**, you can start a task in a number of different ways. **The first task started is the main task. Main task begins subtasks that are also managed by the pool.** Different methods for starting tasks have been discussed below >

- **<T> T invoke(ForkJoinTask<T> task)**

This method starts the **task** and returns the result of the **task**. Calling code waits until method returns.

- **void execute(ForkJoinTask<?> task)**

The execute() method can be used to start a **task** without waiting for its completion.

This method starts the **task**. Calling code continues its execution asynchronously and does not waits for method completion like in invoke method.

- **submit() method comes in 4 different forms.**

submit() method can also be used for submitting task.

- **int getParallelism()**

The method returns **level of parallelism** i.e. number of processors available in the system.

- **void shutdown()**

Initiates shutdown, previously submitted tasks are executed, but no new tasks will be accepted.

- **List<Runnable> shutdownNow()**

- attempts to stop all actively executing tasks,
- submitted tasks may or may not execute.
- awaiting tasks will never execute, and
- method cancels both existing and unexecuted tasks, so it returns empty list.

## ForkJoinTask<V> in java

**ForkJoinTask** is abstract class for tasks that run within a **ForkJoinPool**.

**ForkJoinTask<V>** is an abstract class that defines a task that can be managed by a **ForkJoinPool**.

The **V** specifies the result type of the task in java.

**Threads managed by ForkJoinPool executes ForkJoinTasks.** Small number of threads are used to serve large number of tasks.

### *ForkJoinTask important methods >*

**ForkJoinTask** core methods are **fork( )** and **join()**

- **ForkJoinTask<V>**

**fork( )**

The **fork( )** method **submits the task** for asynchronous execution, means that the thread that calls **fork( )** method to submit task continues to run. Task are executed in the **compute()** method, which is running within a **ForkJoinPool**.

- **V join( )**

The **join( )** method waits for task completion on which it is called. The method returns result of the task.

- **In short, about fork( ) and join( )** are used for starting one or more new tasks and then wait for them to complete.

- **V invoke( )**

**The invoke() method combines the functionality of fork() and join() methods.** **invoke()** submits the task and waits for completion of submitted task.

The method returns result of task.

- **static void invokeAll(ForkJoinTask<?> t1, ForkJoinTask<?> t2)**

**invokeAll()** method submits **t1** and **t2** and waits for completion of **t1** and **t2**.

- **static void invokeAll(ForkJoinTask<?> ... tasks)**

**invokeAll()** method submits list of tasks i.e. **tasks** and waits for completion of all tasks in list.

The ***invokeAll( )*** method can only be called from within the overridden compute() method of another **ForkJoinTask**, which is running within a **ForkJoinPool**.

*Some other important methods for checking status of submitted task -*

- ***boolean isDone()*** method returns true if a task completes.
- ***boolean isCompletedNormally()*** method returns true if a task completed normally without cancellation or without throwing any exception.
- ***boolean isCompletedAbnormally()*** returns true if a task completed abnormally either by cancellation or by throwing any exception.
- ***boolean isCancelled()*** returns true if the task was cancelled.

### **Question 38. Similarity and Difference between RecursiveAction and RecursiveTask in java?**

**Answer.** Another important thread concurrency interview question.

#### **Difference between RecursiveAction and RecursiveTask in java**

| <b><i>RecursiveAction</i></b>                             | <b><i>RecursiveTask&lt;V&gt;</i></b>              |
|-----------------------------------------------------------|---------------------------------------------------|
| This submits a task and does not return a result in java. | This submits a task and returns a result in java. |

|                                                                                 |                                                                                                     |
|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| Definition of compute method<br><b><i>protected abstract void compute()</i></b> | <b><i>protected abstract V compute()</i></b><br>The <b>V</b> specifies the result type of the task. |
|---------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|

### ***Similarity between RecursiveAction and RecursiveTask***

- > Both **extends ForkJoinPool**.
- > All **computations by tasks are performed inside compute() method**.

### ***Question 39. How can we use Fork/Join framework in real world?***

**Answer.** This thread concurrency interview question will test your real time thread concurrency implementation skills. We can use Fork/Join framework for calculating sum of array of 100000 or even may be more numbers. *Fork/Join framework uses divide-and-conquer strategy for enabling parallel programming*. Divide-and-conquer strategy recursively divides a array into smaller subarrays until subarray isn't small enough to be solved independently.

Also, **ForkJoinPool** uses ***work-stealing approach for managing threads***. Each thread in **ForkJoinPool** maintains a queue of tasks. If one thread's queue is empty, it can take task from another thread. This overall improve the programs performance. Please see [program](#) to calculate sum of array of 100000 numbers.

```
/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */
```

### ***Thread concurrency interview Question 40. Difference between synchronized and ReentrantLock in java?***

**Answer.** This is another **very very important** thread concurrency interview question. Fresher and experienced both must be well versed with this.

| <b>synchronized in java</b>                                                                                                                                                                                                                       | <b>ReentrantLock in java</b>                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Does not provide any fair locks in java.</b>                                                                                                                                                                                                   | <p><b>provides fair locks</b>, when lock is fair - first lock is obtained by longest-waiting thread in java.</p> <p>Constructor to provide fairness -</p> <p><b>ReentrantLock(boolean fair)</b></p> <p>Creates an instance of ReentrantLock.</p> <p>When <b>fair</b> is set true, first lock is obtained by longest-waiting thread in java.</p> <p>If <b>fair</b> is set false, any waiting thread could get lock, at discretion of implementation in java.</p> |
| <b>Does not provide tryLock() method or its functionality.</b><br>Thread always waits for lock in java.                                                                                                                                           | <p><b>Provide tryLock() method. If lock is held by another thread then method return false in java.</b></p> <p><b>boolean tryLock()</b></p> <p>Acquires the lock if it is not held by another thread and returns true. <b>And sets lock hold count to 1.</b></p> <p>If current thread already holds lock then method returns true. <b>And increments lock hold count by 1.</b></p> <p>If lock is held by another thread then method return false in java.</p>   |
| There is <b>no method for lock interruptibility</b> , though current thread waits until one of the following thing happens -<br><ul style="list-style-type: none"> <li>• The <b>lock is acquired by the current thread in java</b>, or</li> </ul> | <p><b>void lockInterruptibly()</b></p> <p>If current thread already holds lock then method returns true. <b>And increments lock hold count by 1.</b></p> <p>If the lock is held by another thread then the current thread waits until one of the following thing happens -</p>                                                                                                                                                                                  |

|                                                                                                                                                                |                                                                                                                                                                                                                                                                                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"> <li>Some other thread <b>interrupts the current thread in java.</b></li> </ul>                                              | <ul style="list-style-type: none"> <li>The <b>lock is acquired by the current thread in java</b>, or</li> <li>Some other thread <b>interrupts the current thread.</b></li> </ul> <p>As soon as current thread acquires the lock it <b>sets lock hold count to 1</b>.</p>                                                                                                                              |
| <b>Does not provide any method to return number of threads that may be waiting to acquire this lock in java.</b>                                               | provide <i>int getQueueLength()</i> method to return number of threads that may be waiting to acquire this lock in java.                                                                                                                                                                                                                                                                              |
| <b>holdsLock()</b> method is used to <b>find out whether lock is held by current thread or not</b> . If current thread holds lock method returns true in java. | <i>isHeldByCurrentThread()</i> method is used to <b>find out whether lock is held by current thread or not</b> . If current thread holds lock method returns true in java.                                                                                                                                                                                                                            |
| Thread can hold lock on object monitor only once in java.                                                                                                      | if current thread <b>already holds lock</b> then <b>lock hold count is increased by 1</b> when lock() method is called.<br><br>method to maintain <b>lock hold count</b> -<br><br><b>void lock()</b><br><br>Acquires the lock if it is not held by another thread. <b>And sets lock hold count to 1</b> .<br><br>If current thread already holds lock then <b>lock hold count is increased by 1</b> . |
| Does not provide any new condition() method in java.                                                                                                           | provides <i>newCondition()</i> method.<br><br>Method returns a Condition instance to be used with this Lock instance.<br><br>Condition instance are similar to using <u><a href="#">Wait()</a></u> , <u><a href="#">notify()</a></u> and <u><a href="#">notifyAll()</a></u> methods on object.                                                                                                        |

|  |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
|--|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <ul style="list-style-type: none"> <li>• <b>IllegalMonitorStateException</b> is thrown if this lock is not held when any of the <b>Condition waiting or signalling methods</b> are called.</li> <li>• <b>Lock is released</b> when the <b>condition waiting methods are called</b> and before they return, the lock is reacquired and the <b>lock hold count</b> restored to what it was when the method was called.</li> <li>• If a <b>thread is interrupted while waiting</b> then <b>InterruptedException</b> will be thrown and following things will happen - <ul style="list-style-type: none"> <li>◦ the <b>wait will be over</b>, and</li> <li>◦ <b>thread's interrupted status will be cleared</b>.</li> </ul> </li> <li>• Waiting threads are signalled in FIFO (first in first out order) order in java.</li> <li>• When lock is <b>fair</b>, first lock is obtained by longest-waiting thread in java.<br/>If lock is not <b>fair</b>, any waiting thread could get lock, at discretion of implementation in java.</li> </ul> |
|--|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## Question 41. Difference between traditional multithreading and parallel programming in java?

**Answer.** Basic interview question. Not a important one. MultiThreading primarily was designed to work with single CPU and utilize idle time of CPU. If two or more processors are there multithreading won't be able to utilize multi processors but parallel programing using Fork/Join framework can utilize multiple processors available in computer in java.

## Question 42. Explain atomic operations in java?

8. Answer. Developers must have knowledge of atomic operations in thread concurrency java. Java provides some classes in `java.util.concurrent.atomic` which offers an alternative to the other synchronization in java.
9. Please see [Atomic operations in java.](#)

## Question 43. What is AtomicInteger in java? Explain key methods of AtomicInteger?

**Answer.** Another very important thread concurrency interview question for freshers and experienced. *AtomicInteger provides you with **int value** that is updated atomically. i.e. we can use these classes without any explicit synchronization in multithreading environment, because any operation done on these classes is thread safe in java.*

### AtomicInteger important Methods >

- **`int get()`**  
method returns the current value
- **`void set(int newValue)`**  
Sets to **newValue**.
  
  
  
- **`int getAndSet(int newValue)`**  
Sets to **newValue** and returns the old value.
  
  
  
- **`boolean compareAndSet(int expect, int update)`**  
Compare with **expect**, if equal, set to **update** and return true.

### *Addition methods >*

- *int addAndGet(int value)*

adds **value** to the current value. And **return updated value**.

- *int incrementAndGet()*

increments current value by 1. And **return updated value**.

- *int getAndAdd(int value)*

Method **return current value**. And adds **value** to the current value.

- *int getAndIncrement()*

Method **return current value**. And increments current value by 1.

### *Subtraction methods >*

- *int decrementAndGet()*

decrements current value by 1. And **return updated value**.

- *int getAndDecrement()*

Method **return current value**. And decrements current value by 1.

## Thread concurrency interview Question 44. How can you implement your own AtomicInteger in java?

**Answer.** Another important and complex interview question for developers. Please see [Implementation of custom/own AtomicInteger in java](#).

## Question 45. What is AtomicLong? Explain key methods of AtomicLong in java?

**Answer.** This is another important thread concurrency interview question for freshers and experienced. AtomicLong provides you with **long value** that is updated atomically. i.e. we can use these classes without any explicit synchronization in multithreading environment, because any operation done on these classes is thread safe.

### AtomicLong important Methods >

- **long get()**

method returns the current value

- **void set(long newValue)**

Sets to **newValue**.

- **long getAndSet(long newValue)**

Sets to **newValue** and returns the old value.

- **boolean compareAndSet(long expect, long update)**

Addition methods >

- **long addAndGet(long value)**

adds **value** to the current value. And **return updated value**.

- **long incrementAndGet()**

increments current value by 1. And **return updated value**.

- **long getAndAdd(long value)**

Method **return current value**. And adds **value** to the current value.

- **long getAndIncrement()**

Method **return current value**. And increments current value by 1.

### Subtraction methods >

- ***long decrementAndGet()***  
decrements current value by 1. And **return updated value.**
- ***long getAndDecrement()***  
Method **return current value.** And decrements current value by 1.

## Thread concurrency interview Question 46. How can you implement your own AtomicLong in java?

**Answer.** Another important and complex interview question for developers. Please see [Implementation of custom/own AtomicLong in java.](#)

`/** Copyright (c), AnkitMittal JavaMadeSoEasy.com */`

## Thread concurrency interview Question 47. What will be the output of below question in java?

```

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ReentrantLockTryLockTest {
 public static void main(String[] args) {

 Lock lock=new ReentrantLock();
 MyRunnable myRunnable=new MyRunnable(lock);
 new Thread(myRunnable,"Thread-1").start();
 new Thread(myRunnable,"Thread-2").start();

 }
}

class MyRunnable implements Runnable{

 Lock lock;
 public MyRunnable(Lock lock) {
 this.lock=lock;
 }

 public void run(){

 System.out.println(Thread.currentThread().getName()
 +" is Waiting to acquire lock");

 if(lock.tryLock()){
 System.out.println(Thread.currentThread().getName()
 +" has acquired lock.");
 try {
 Thread.sleep(1000);
 } catch (InterruptedException e) {
 e.printStackTrace();
 }
 }
 else{
 System.out.println(Thread.currentThread().getName()
 +" didn't get lock.");
 }
 }
}

```

## Answer.

Thread-1 is Waiting to acquire lock

Thread-2 is Waiting to acquire lock

Thread-1 has acquired lock.

Thread-2 didn't got lock.

## Thread concurrency interview Question 48. What will be the output of below question in java?

```
import java.util.concurrent.Phaser;

public class PhaserParentChildTest {
 public static void main(String[] args) {

 /*
 * Creates a new phaser with no registered unArrived parties.
 */
 Phaser parentPhaser = new Phaser();

 /*
 * Creates a new phaser with the given parent &
 * no registered unArrived parties.
 */
 Phaser childPhaser = new Phaser(parentPhaser,0);

 childPhaser.register();

 System.out.println("parentPhaser isTerminated : "+parentPhaser.isTerminated());
 System.out.println("childPhaser isTerminated : "+childPhaser.isTerminated());

 childPhaser.arriveAndDeregister();
 System.out.println("\n--childPhaser has called arriveAndDeregister()-- \n");

 System.out.println("parentPhaser isTerminated : "+parentPhaser.isTerminated());
 System.out.println("childPhaser isTerminated : "+childPhaser.isTerminated());

 }
}
```

### Answer.

parentPhaser isTerminated : false

```
childPhaser isTerminated : false
--childPhaser has called arriveAndDeregister()--
parentPhaser isTerminated : true
childPhaser isTerminated : true
```

## Thread concurrency interview Question 49.

**Which atomic classes are available and which are not available in java 7? And why jdk developers didn't created those classes?**

**Answer.** This is confusing thread concurrency interview question for freshers.

Java provides following classes in [java.util.concurrent.atomic](#) >

*Classes found in [java.util.concurrent.atomic](#) are >*

- **AtomicInteger,**
- **AtomicLong, and**
- **AtomicBoolean.**

*Classes NOT found in [java.util.concurrent.atomic](#) are >*

- **AtomicByte,**
- **AtomicShort,**
- **AtomicFloat,**
- **AtomicDouble and**
- **AtomicCharacter**

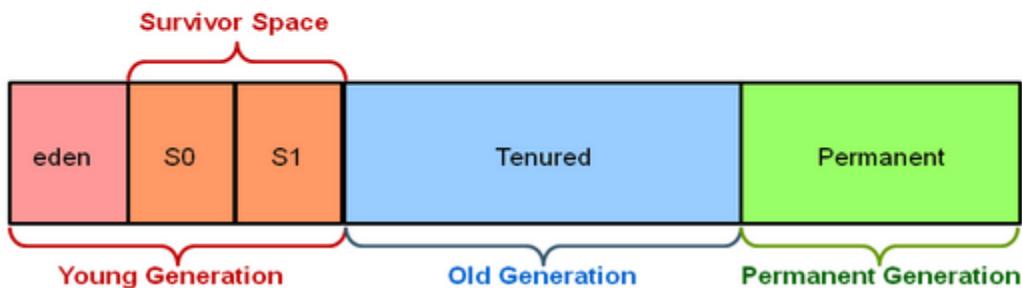


# Garbage Collection

1. Give digramatic JVM Heap memory (Hotspot heap structure).

JVM Heap memory (Hotspot heap structure) with diagram in java >

## Hotspot Heap Structure



2. What is Throughput in gc(garbage collection) in java ?

In short, Throughput is the **time not spent** in garbage collection (GC) (in percent).

Throughput focuses on maximizing the amount of work by an application in a specific period of time. Examples of how throughput might be measured include >

- The number of transactions completed in a given time.
- The number of jobs that a batch program can complete in an hour.
- The number of database queries that can be completed in an hour.

3. What are pauses in gc(garbage collection) in java?

Pauses is applications pauses i.e. when **application doesn't give any response** because of garbage collection (GC).

**4. JVM Heap memory (Hotspot heap structure) in java consists of which elements?**

1. **Young Generation**
  - 1a) **Eden,**
  - 1b) **S0 (Survivor space 0)**
  - 1c) **S1 (Survivor space 1)**
2. **Old Generation (Tenured)**
3. **Permanent Generation.**

So, JVM Heap memory (Hotspot heap structure) is divided into three parts **Young Generation**, **Old Generation (tenured)** and **Permanent Generation**.

**Young Generation** is further divided into **Eden**, **S0 (Survivor space 0)** and **S1 (Survivor space 1)**.

**5. What is Young Generation (Minor garbage collection occurs in Young Generation)?**

New objects are allocated in Young generation. **Minor garbage collection occurs in Young Generation.**

**When minor garbage collection?**

When the young generation fills up, this causes a **minor garbage collection**.

All the unreferenced (dead) objects are cleaned up from young generation.

**When objects are moved from young to old generation in JVM heap?**

Some of the objects which aren't cleaned up **survive in young generation and gets aged**. Eventually such objects are **moved from young to old generation**.

**What is Stop the World?**

Minor garbage collections are called **Stop the World** events. **All the non-daemon threads running in application are stopped during minor garbage collections** (i.e. the application stops for while).

**Daemon threads performs minor garbage collection.** (Daemon threads are low priority threads which runs intermittently in background for doing garbage collection).

- 1a) **Eden,**

- 1b) **S0 (Survivor space 0)**
- 1c) **S1 (Survivor space 1)**

6. **What is Old Generation or (tenured generation) - (Major garbage collection occurs in Old Generation)?**

The **Old Generation** is used for storing the long surviving aged objects (Some of the objects which aren't cleaned up **survive in young generation and gets aged**. Eventually such objects are **moved from young to old generation**).

**Major garbage collection occurs in Old Generation.**

**At what time (or what age) objects are moved from young to old generation in JVM heap?**

There is some threshold set for young generation object and when that age is met, the object gets moved to the old generation during gc(garbage collection) in java.

**What is major garbage collection in java?**

When the old generation fills up, this causes a **major garbage collection**. Objects are cleaned up from old generation. Major collection is much slower than minor garbage collection in jvm heap **because it involves all live objects**.

**Major garbage collection are Stop the World Event in java?**

Major garbage collections are also called Stop the World events. All the non-daemon threads running in application are stopped during major garbage collections. [Daemon threads](#) performs major garbage collection.

**Major gc(garbage collections) in responsive applications in java?**

Major garbage collections should be minimized for responsive applications because applications must not be stopped for long.

**Optimizing Major gc(garbage collections) in responsive applications in java?**

**Selection of appropriate garbage collector for the old generation** space affects the length of the "Stop the World" event for a major garbage collection.

7. **What is Permanent Generation or (Permgen) - (full garbage collection occurs in permanent generation in java)?**

Permanent generation Space contains metadata required by JVM to describe the classes and methods used in the application.

The permanent generation is included in a **full garbage collection** in java. The permanent generation space is populated at runtime by JVM based on classes in use in the application.

The permanent generation space also contains **Java SE library classes and methods** in java.

JVM garbage collects those classes when classes are no longer required and space may be needed for other classes in java.

## 8. What are the most important VM (JVM) PARAMETERS in JVM Heap memory?

**-Xms** : Xms is **minimum heap size** which is allocated at initialization of JVM.

**Examples** of using **-Xms** VM (JVM) option in java >

Example1 of using **-Xms** VM (JVM) option in java >

```
java -Xms512m MyJavaProgram
```

It will set the minimum heap size of JVM to 512 megabytes.

Example2 of using **-Xms** VM (JVM) option in java >

```
java -Xms1g MyJavaProgram
```

It will set the minimum heap size of JVM to 1 gigabyte.

**-Xmx** : Xmx is the **maximum heap size** that JVM can use.

**Examples** of using **-Xmx** VM option in java >

Example1 of using **-Xmx** VM (JVM) option in java >

```
java -Xmx512m MyJavaProgram
```

It will set the maximum heap size of JVM to 512 megabytes.

Example2 of using **-Xmx** VM (JVM) option in java >

```
java -Xmx1g MyJavaProgram
```

It will set the maximum heap size of JVM to 1 gigabyte.

## 9. What are parameters for Young Generation(VM PARAMETERS for Young Generation) ?

**-Xmn :** -Xmn sets the size of young generation.

**Examples** of using **-Xmn** VM (JVM) option in java >

Example1 of using **-Xmn** VM (JVM) option in java >

```
java -Xmn512m MyJavaProgram
```

Example2 of using **-Xmn** VM (JVM) option in java >

```
java -Xmn1g MyJavaProgram
```

**-XX:NewRatio :** NewRatio controls the size of young generation.

**Example** of using **-XX:NewRatio** VM option in java >

**-XX:NewRatio=3 means that the ratio between the young and old/tenured generation is 1:3.**

In other words, the combined size of the eden and survivor spaces will be one fourth of the total heap size.

**-XX:NewSize** - NewSize is **minimum size of young generation** which is allocated at initialization of JVM.

Note : If you have specified **-XX:NewRatio** than minimum size of the young generation is allocated automatically at initialization of JVM.

**-XX:MaxNewSize** - MaxNewSize is the **maximum size of young generation** that JVM can use.

- 1a) **Eden**,
- 1b) **S0 (Survivor space 0)**
- 1c) **S1 (Survivor space 1)**

**-XX:SurvivorRatio : (for survivor space)**

SurvivorRatio can be used to **tune the size of the survivor spaces**, but this is often not as important for performance.

Example of using **-XX:SurvivorRatio** > **-XX:SurvivorRatio=6 sets the ratio between each survivor space and eden to be 1:6.**

In other words, each survivor space will be one eighth of the young generation (not one seventh, because there are two survivor spaces).

**10. What are the parameters for the old Generation (tenured) - (VM PARAMETERS for Old Generation) ?**

**-XX:NewRatio :** NewRatio controls the size of young and old generation.

**Example of using -XX:NewRatio, -XX:NewRatio=3 means that the ratio between the young and old/tenured generation is 1:3.** In other words, the combined size of the eden and survivor spaces will be one fourth of the total heap size.

**11. What are the parameters for Permanent Generation (VM PARAMETERS for Permanent Generation)?**

**-XX:PermSize:** It's initial value of Permanent Space which is allocated at startup of JVM.

**Examples of using -XX:PermSize VM (JVM) option in java >**

Example1 of using **-XX:PermSize** VM (JVM) option in java >

java -XX:PermSize=512m MyJavaProgram

It will set initial value of Permanent Space as 512 megabytes to JVM

Example2 of using **-XX:PermSize** VM (JVM) option in java >

java -XX:PermSize=1g MyJavaProgram

It will set initial value of Permanent Space as 512 gigabyte to JVM

**-XX:MaxPermSize:** It's maximum value of Permanent Space that JVM can allot up to.

**Examples of using -XX:MaxPermSize VM option in java >**

Example1 of using **-XX:MaxPermSize** VM (JVM) option in java >

java -XX:MaxPermSize=512m MyJavaProgram

It will set maximum value of Permanent Space as 512 megabytes to JVM

Example2 of using **-XX:MaxPermSize** VM (JVM) option in java >

`java -XX:MaxPermSize=1g MyJavaProgram`

It will set maximum value of Permanent Space as 1 gigabyte to JVM

**12. Name the other important VM (JVM) parameters for java heap in java.**

**-XX:MinHeapFreeRatio and -XX:MaxHeapFreeRatio**

JVM can grows or shrinks the heap to keep the proportion of free space to live objects within a specific range.

**-XX:+AggressiveHeap** is used for Garbage Collection Tuning setting. This VM option inspects the server resources and attempts to set various parameters in optimal manner for long running and memory consuming applications. There must be minimum of 256MB of physical memory on the servers before the AggressiveHeap can be used.

**-Xss** > Use this VM option to **adjust the maximum thread stack size**.

Also you must know that -Xss option is same as **-XX:ThreadStackSize**.

**Examples of using -Xss VM option in java >**

Example1 of using -Xss >

`java -Xss512m MyJavaProgram`

It will set the default stack size of JVM to 512 megabytes.

Example2 of using -Xss >

`java -Xss1g MyJavaProgram`

It will set the default stack size of JVM to 1 gigabyte.

**13. What are the different Garbage collectors in java?**

- I. Serial collector / Serial GC (Garbage collector)
- II. Throughput GC (Garbage collector) or Parallel collector
- III. Incremental low pause garbage collector (train low pause garbage collector)
- IV. Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector
- V. G1 Garbage Collector (or Garbage First)

**14. What is Serial collector / Serial GC (Garbage collector) ?**

**Features of Serial GC (Garbage collector) in java :**

Serial collector is also called Serial GC (Garbage collector) in java.

Serial collector is simply also called Serial collector in java.

Serial GC (Garbage collector) is rarely used in java.

Serial GC (Garbage collector) is designed for the single threaded environments in java.

In Serial GC (Garbage collector) , both minor and major garbage collections are done serially by one thread (using a single virtual CPU) in java.

Serial GC (Garbage collector) uses a mark-compact collection method. This method moves older memory to the beginning of the heap so that new memory allocations are made into a single continuous chunk of memory at the end of the heap. This compacting of memory makes it faster to allocate new chunks of memory to the heap in java.

The serial garbage collector is the default for client style machines in Java SE 5 and 6.

***When to Use the Serial GC (garbage Collector) in java?***

The Serial GC is the garbage collector of choice for most applications that do not have low pause time requirements and run on client-style machines. It takes advantage of only a single virtual processor for garbage collection work in java.

Serial GC (garbage collector) is also popular in environments where a high number of JVMs are run on the same machine. In such environments when a JVM does a garbage collection it is better to use only one processor to minimize the interference on the remaining JVMs in java.

***Vm (JVM) option for enabling serial GC (garbage Collector) in java.***

-XX:+UseSerialGC

Example of Passing Serial GC in Command Line for starting jar>

java -Xms256m -Xms512m -XX:+UseSerialGC -jar d:\MyJar.jar

**15. What is Throughput GC (Garbage collector) or Parallel collector in java?**

### ***Features of Throughput GC (Garbage collector) in java .***

- [Throughput collector](#) is also called
- Throughput GC (garbage collector)
- ParallelGC (garbage collector)
- Throughput collector
- ParallelGC collector
- Throughput garbage collector is the default garbage collector for JVM in java.
- Throughput garbage collector uses multiple threads to execute a minor collection and so reduces the serial execution time of the application in java.
- The throughput garbage collector is similar to the serial garbage collector but uses multiple threads to do the minor collection in java.
- This garbage collector uses a parallel version of the young generation garbage collector in java.
- The tenured generation collector is the same as the serial garbage collector in java.

### ***When to Use the Throughput GC (Garbage collector) in java ?***

- The Throughput garbage collector should be used when application can afford low pauses in java.
- And application is running on host with multiple CPU's (to derive advantage of using multiple threads for garbage collection) in java.

### ***Vm (JVM) option for enabling throughput GC (Garbage collector) in java .***

**-XX:+UseParallelGC**

**Example of using throughput collector in Command Line for starting jar>**

```
java -Xms256m -Xms512m -XX:+UseParallelGC -jar d:\MyJar.jar
```

With this **Vm** (JVM) option you get a

- **Multi-threaded young** generation garbage collector in java,
- **single-threaded old** generation garbage collector in java and

- **single-threaded compaction** of old generation in java.

**Vm** (JVM) option for enabling **throughput collector** with **n number of threads** in java >

-XX:ParallelGCThreads=<numberOfThreads>

**Another Vm** (JVM) option for enabling **throughput collector** in java >

-XX:+UseParallelOldGC

#### 5.2.4. Goals for Throughput GC (Garbage collector) in java >

- Maximum pause time goal (Highest priority)
- Throughput goal
- Minimum footprint goal (Lowest priority)

**Performance of Throughput GC (garbage Collector) host with different number of CPU's in java**

**Vm** (JVM) option for enabling **throughput collector** with **n number of threads** in java >

**Another Vm** (JVM) option for enabling **throughput collector** in java >

**Controlling maximum pause time and throughput** for the application in java >

**Vm** (JVM) option for **maximum pause time** in java >

**Vm** (JVM) option for **throughput** in java >

Adjusting Generation Sizes in throughput GC (Garbage collector).

## 16. What is Incremental low pause garbage collector (train low pause garbage collector) in java?

We won't be discussing in detail about incremental low pause garbage collector because is **not used these days** in java. Incremental low pause collector was used in Java 4.

Vm (JVM) option which was used for enabling Incremental low pause garbage collector in java >

-XX:+UseTrainGC.

## 17. What is Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java?

*Features of Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java*

[Concurrent Mark Sweep Collector](#) is also called concurrent low pause collector

concurrent low pause GC (garbage collector). Concurrent Mark Sweep (CMS) collector **collects the old/tenured generation** in java.

Concurrent Mark Sweep (CMS) Collector **minimize the pauses** by doing most of the **garbage collection work concurrently with the application threads** in java.

Concurrent Mark Sweep (CMS) Collector **on live objects**. Concurrent Mark Sweep (CMS) Collector **does not copy or compact the live objects**. A garbage collection is done **without moving the live objects**. If fragmentation becomes a problem, allocate a larger heap in java.

*When to Use the Concurrent Low Pause Collector in java*

Concurrent Low Pause Collector should be used if your **applications that require low garbage collection pause times** in java.

Concurrent Low Pause Collector should be used when your **application can afford to share processor resources with the garbage collector while the application is running** in java.

Concurrent Low Pause Collector is beneficial to applications which have a relatively **large set of long-lived data** (a large tenured generation) and run on machines with **two or more processors** in java.

**Examples** when to use Concurrent Mark Sweep (CMS) collector / concurrent low pause collector should be used for >

**Example 1 - Desktop UI application** that **respond to events**,

**Example 2 - Web server responding to a request** and

**Example 3 - Database responding to queries.**

*Vm (JVM) option for enabling Concurrent Mark Sweep (CMS) Collector in java.*

**-XX:+UseConcMarkSweepGC**

**Example** of using Concurrent Mark Sweep (CMS) collector / **concurrent low pause collector** in Command Line for starting jar>

```
java -Xms256m -Xms512m -XX:+UseConcMarkSweepGC -jar d:\MyJar.jar
```

*Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector working in detail in java*

As mentioned above Concurrent Mark Sweep (CMS) collector **collects the old/tenured generation (i.e. performs Major garbage collection process)**.

*Major gc(garbage collection) in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java.*

**For each major collection** the CMS collector will **pause all the application threads for a brief period** at the **beginning** of the collection and toward the **middle** of the collection.

The **second pause** tends to be the **longer** than first pause and **uses multiple threads to do the collection** work during that pause in java. The remainder of the collection is done with a garbage collector thread that runs concurrently with the application.

*Minor gc (garbage collection) in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector.*

The minor collections is done in a manner **similar to the serial collector** although **multiple threads are used** to do the collection in java.

### *Heap Structure for CMS garbage Collector*

CMS garbage collectors divides heap into three sections: **young** generation, **old** generation, and **permanent** generation of a fixed memory size. **Young Generation** is further divided into **Eden**, **S0 (Survivor space 0)** and **S1 (Survivor space 1)**.

#### **Young Generation GC** (garbage Collection) in java

**Live objects** are copied from the **Eden space and survivor space** to the **other survivor space**.

Any **older objects** that have reached their aging threshold are **promoted to old generation**.

After a young GC, the **Eden space and one of the survivor spaces is cleared**.

promoted objects (**older objects** that have reached their aging threshold in young GC) are available in **old generation**.

#### **Old Generation GC** (garbage Collection) with CMS in java

**Initial mark phase** - (**First pause** happens/ stop the world event ) - **mark live/reachable objects** (Example - objects on thread stack, static objects etc.) and elsewhere in the **heap** (Example - the young generation).

**Concurrent marking** phase - (No pause phase ) - finds **live objects** while the application continues to execute.

**Remark** - (**Second pause** happens/ stop the world events) - It finds **objects** that were missed during the concurrent marking phase due to the concurrent execution of the application threads.

Old Generation GC (garbage Collection) - **Sweep phase** (Concurrent Sweep phase) in java

**Sweep** phase - do the concurrent **sweep**, memory is freed up.

Objects that were not marked in the previous phase are deallocated in place.

There is no compaction. **Unmarked objects** are equal to **Dead Objects**.

#### Old Generation GC (garbage Collection) - After Sweeping

**Reset** phase - do the concurrent **reset**.

## 18. What is G1 Garbage Collector (or Garbage First) in java?

The G1 garbage collector **features** -

- [G1 garbage collector](#) is also called
- G1 garbage collector
- G1 collector
- G1 GC (garbage collector)
- Garbage first collector
- G1 garbage collector was **introduced in Java 7**
- G1 garbage collector was designed to replace CMS collector(Concurrent Mark-Sweep garbage Collector).
- G1 garbage collector is **parallel**,
- G1 garbage collector is **concurrent**, and
- G1 garbage collector is **incrementally compacting low-pause** garbage collector in java.
- G1 garbage collector has much better layout from the other garbage collectors like serial, throughput and CMS garbage collectors in java.
- G1(Garbage First) collector **compacts sufficiently to completely avoid the use of fine-grained free lists for allocation**, and instead relies on regions.
- G1(Garbage First) collector **allows customizations** by allowing users to specify pause times.
- G1 Garbage Collector (or Garbage First) limits **GC pause times and maximizes throughput**.

**Vm** (JVM) option for enabling G1 Garbage Collector (or Garbage First) in java >

**-XX:+UseG1GC**

Example of using G1 Garbage Collector in Command Line for starting jar>

java -Xms256m -Xms512m **-XX:+UseG1GC** -jar d:\MyJar.jar

G1(Garbage First) collector functioning >

CMS garbage collectors divides heap into three sections: young generation, old generation, and permanent generation of a fixed memory size. All memory objects end up in one of these three sections. The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory. The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions), **but** there is not a

fixed size for them. This provides **greater flexibility in memory usage**.

### **When to use G1 garbage collector ?**

G1 must be used when applications that require **large heaps** with limited GC latency.

Example - Application that require

- **heaps around 5-6GB or larger** and
- **pause time required below 0.5 seconds**

### **When to switch from CMS (or old garbage collectors) to G1 garbage collector?**

Applications using CMS garbage collector may switch to G1 when >

- **Full GC** durations are too **long** or too **frequent**.
- The **rate of object allocation or promotion varies** significantly.
- **Long garbage collection** (longer than 0.5 to 1 second)

### **The G1(Garbage First) collector working Step by Step?**

The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.

G1(Garbage First) garbage collector Heap Structure

The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions), but there is not a fixed size for them. This provides greater flexibility in memory usage. Each **region's size** is chosen by **JVM at startup**. Generally heap is divided into **2000 regions** by JVM varying in size from **1 to 32Mb**.

G1(Garbage First) garbage collector Heap Allocation

As mentioned above there are following region in heap >**Eden, survivor** and **old** generation region. Also, **Humongous and unused** regions are there in heap.

Young Generation in G1 garbage collector

Generally heap is divided into **2000 regions** by **JVM**. **Minimum** size of region can be **1Mb** and **Maximum** size of region can be **32Mb**. Regions are not required to be contiguous like CMS garbage collector.

Young GC in G1 garbage collector

- **Live objects** are copied or moved to **survivor regions**.
- If objects aging threshold is met it get promoted to **old** generation regions.

- It is **STW** (stop the world) event. Eden size and survivor size is calculated for the next young GC.
- The young GC is done parallelly using multiple threads.

End of a Young GC with G1 garbage collector At this stage **Live objects have been evacuated (copied or moved) to >**

- **survivor** regions or
- **old** generation regions.

Old Generation Collection with G1 garbage collector

G1 collector is low pause collector for old generation objects.

Initial Mark -

- It is **STW** (stop the world) event.
- With G1, it is **piggybacked on a normal young GC**. Mark survivor regions (root regions) which may have references to objects in old generation.

Root Region Scanning -

- **Scan survivor regions for references into the old generation.**
  - This happens while the **application continues to run**. The phase must be **completed before** a young GC can occur.

Concurrent Marking -

- **Find live objects over the entire heap.**
- This happens while the **application is running**.
- This phase can be interrupted by young generation garbage collections.

Remark (Stop the World Event) -

- **Completes the marking of live object in the heap.**
- Uses an algorithm called **snapshot-at-the-beginning** (SATB) which is much **faster** than algorithm used in the **CMS** collector.

Cleanup (Stop the World Event and Concurrent) -

- **Performs accounting on live objects and completely free regions.** (Stop the world)
  - Young generation and old generation are reclaimed at the same time
  - Old generation regions are selected based on their liveness.
  
- **Scrubs** the Remembered Sets. (Stop the world)
- **Reset the empty regions** and return them to the free list. (Concurrent)

**19. Difference between Serial GC (Garbage collector) vs Throughput GC (Garbage collector) in java?**

**Difference Serial and Throughput gc (garbage Collector).**

Serial collector **uses one thread to execute garbage collection.**

Throughput collector **uses multiple threads to execute garbage collection.**

Serial GC is the garbage collector of choice for applications that do **not have low pause time requirements** and run on client-style machines. Throughput GC is the garbage collector of choice for applications that have **low pause time requirements**.

**20. What is ParNew collector ?**

Is the young generation collector. It is the parallel copy collector, it uses multiple threads in parallel. Vm parameter for enabling ParNew collector is -XX:+UseParNewGC.

## 21. What is Automatic Garbage Collection in JVM heap memory in java?

**Automatic garbage collection** is the process of

- **Identifying objects which are in use** in java heap memory and
- **Which objects are not in use** in java heap memory and
- **deleting the unused objects** in java heap memory.

*How to Identify objects which are in use in JVM heap memory in java?*

**Objects** in use (or **referenced objects**) are those objects which are still needed by java program, some part of java program is still pointing to that object.

*Which objects are not in use in JVM heap memory in java?*

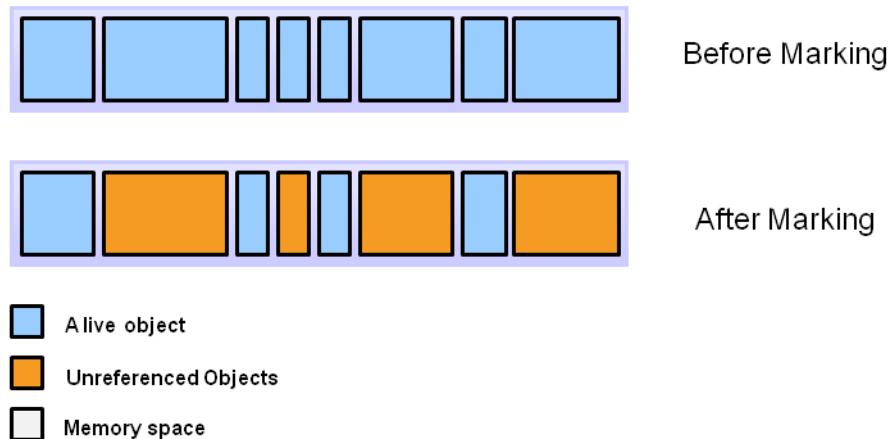
**Objects not** in use (or **unreferenced objects**) are those objects which are not needed by java program, no part of java program is pointing to that object. So, these unused objects can be cleaned in GC (garbage collection) process and memory used by an unreferenced object can be reclaimed.

## 22. How garbage collection is done using Marking and deletion in java?

### 7.1) Step 1 > **Marking**

Marking is a process in which gc (garbage collector) identifies which parts of memory (occupied by objects) are in use and which are not.

## Marking



Before Marking >

All the objects are shown in **blue**, at this stage

- some of objects might be **in use** (referenced objects) and
- some of objects might **not be in use** (unreferenced objects) .

After Marking >

**Objects in use (or referenced objects or Alive objects)** are shown in **blue**.

**Objects not in use (or unreferenced objects)** objects are shown in **Orange**.

### 7.2) Step 2 > Deletion

#### Step 2a : Normal Deletion

- **Normal deletion removes all the unreferenced objects and leaves referenced objects and pointers to free space.**

### **Step 2b : Deletion with Compacting**

Deletion with Compacting is done to improve the performance than normal deleting.

Deletion with Compacting is done to improve the performance than normal deleting.

- Deletion with Compacting **removes all the unreferenced objects and**
- **compacts the remaining referenced objects by moving all the referenced objects together.**
- As all the referenced objects are moved together **new memory allocation becomes easier and much faster.**

### **23. Very important points about GC (Garbage Collection) in Java**

1. All Java objects are always created on heap in java.

#### **2. What is GC (Garbage collection) process in java?**

GC (Garbage collection) is the process by which JVM cleans objects (unused objects) from heap to reclaim heap space in java.

OR

#### **What is Automatic Garbage Collection in JVM heap memory in java?**

**Automatic garbage collection** is the process of

- **Identifying objects which are in use in java heap memory and**
- **Which objects are not in use in java heap memory and**
- **deleting the unused objects in java heap memory.**

#### **3. How to Identify objects which are in use in java heap memory?**

**Objects** in use (or **referenced objects**) are those objects which is still needed by java program, some part of java program is still pointing to that object.

#### **4. Which objects are NOT in use in java heap memory?**

**Objects not** in use (or **unreferenced objects** or **unused objects**) are those objects which is not needed by java program, no part of java program is pointing to that object. So, these unused objects can be cleaned in garbage collection process and memory used by an unreferenced object can be reclaimed.

5. GC (Garbage collection) process **automatically clears objects from heap to reclaim heap space**. You just need to specify the type of garbage collector type you want to use at JVM startup.
6. Gc (garbage collector) **calls finalize method for garbage collection**. **finalize method is called only once by garbage collector for an object in java**.
7. **Daemon threads** are low priority threads which **runs intermittently in background** for doing **garbage collection (gc)** in java.
8. We can **force early gc (garbage collection) in java** by using following methods >

```
System.gc();
Runtime.getRuntime().gc();
```

```
System.runFinalization();
Runtime.getRuntime().runFinalization();
```

9. By calling these methods JVM runs the finalize() methods of any objects pending finalization i.e. objects which have been discarded but their finalize method is yet to be run. After finalize method is executed JVM reclaims space from all the discarded objects in java.

**Note :** Calling these methods does **not guarantee** that it will **immediately start performing garbage collection**.

10. **Finalize method** execution is not assured - We must not override finalize method to write some critical code of application because methods execution is not assured. Writing some critical code in finalize method and relying on it may make application to go horribly wrong in java.

11. Dealing with **OutOfMemoryError** in java.

12. **WeakHashMap** in java - java.util.WeakHashMap is hash table based implementation of the **Map** interface, with **weak keys**.

**An entry in a WeakHashMap will be automatically removed by garbage collector when its key is no longer in ordinary use.** Read in detail about WeakHashMap.

13. Object which is set explicitly set to **null** becomes **eligible for gc** (garbage collection) in java .

Example 1 >

String s="abc"; //s is currently **not eligible** for gc (garbage collection) in java.

s = null; //Now, s is currently **eligible** for gc (garbage collection) in java.

### Example 2 >

List list =new ArrayList(); //list is currently **not eligible** for gc (garbage collection).

list = null; //Now, list is currently **eligible** for gc (garbage collection).

14. **Difference in garbage collection in C/C++ and Java** (Hint : In terms of memory allocation and deallocation of objects)?

In java garbage collection (memory allocation and deallocation of objects) is an **automatic** process.

But, In C and C++ memory allocation and deallocation of objects) is a **manual** process.

15. All the **variables** declared **inside block** becomes **eligible for gc** (garbage collection) **when** we **exit** that **block** (As scope of those variable is only that block) in java.

Example of garbage collection while using block in java -

```
class MyClass {
 public static void main(String[] args) {
 boolean var = false;
 if (var) { // begin block 1
 int x = 1; // x is declared inside block
 //.....
 //code inside block...
 //.....
 } // end block 1 //And now x is eligible for gc (garbage
collection)
 else { // begin block 2
 int y = 1;
 //.....
 //code inside block...
 //.....
 }
 }
}
```

```

 } // end block 2 //And now y is eligible for gc (garbage
collection)

}
}

```

#### 24. Summary of garbage collection in java-

- 1) Terms frequently used in Garbage Collection (GC) in java-

- What is Throughput in gc(garbage collection) in java ?

Throughput is the time not spent in garbage collection (GC) ( in percent).

- What are pauses in gc(garbage collection) in java ?

Pauses is applications pauses when application is paused because of garbage collection (GC).

- 2) JVM Heap memory (Hotspot heap structure) with diagram in java >

- 2.1) JVM Heap memory (Hotspot heap structure) in java consists of following elements>

JVM Heap memory (Hotspot heap structure) is divided into three parts **Young Generation**, **Old Generation** (tenured) and **Permanent Generation**.

**Young Generation** is further divided into **Eden**, **S0 (Survivor space 0)** and **S1 (Survivor space 1)**.

- 3) GARBAGE COLLECTION (Minor and major garbage collection) in JVM Heap memory (i.e. in young, old and permanent generation) >

- 3.1) Young Generation (*Minor garbage collection occurs in Young Generation*)

New objects are allocated in Young generation. When the young generation fills up, this causes a **minor garbage collection**. All the unreferenced (dead) objects are cleaned up from young generation.

- 3.2) Old Generation or (tenured generation) - (*Major garbage collection occurs in Old Generation*)

The **Old Generation** is used for storing the long surviving aged objects. When the old generation fills up, this causes a **major garbage collection**. Objects are cleaned up from old generation.

- **3.3) Permanent Generation or (Permgen) - (full garbage collection occurs in permanent generation in java).**

**Permanent generation Space contains metadata required by JVM to describe the classes and methods used in the application.**

The permanent generation is included in a **full garbage collection** in java.

- **4) Most important VM (JVM) PARAMETERS in JVM Heap memory >**

**-Xms** : Xms is **minimum heap size** which is allocated at initialization of JVM.

**-Xmx** : Xmx is the **maximum heap size** that JVM can use.

- **4.1) Young Generation(VM PARAMETERS for Young Generation)**

**-Xmn** : -Xmn sets the size of young generation.

**-XX:NewRatio** : NewRatio controls the size of young generation.

**-XX:NewSize** - NewSize is **minimum size of young generation** which is allocated at initialization of JVM.

**-XX:MaxNewSize** - MaxNewSize is the **maximum size of young generation** that JVM can use.

**-XX:SurvivorRatio** : **(for survivor space)** SurvivorRatio can be used to **tune the size of the survivor spaces**.

- **4.2) Old Generation (tenured) - (VM PARAMETERS for Old Generation)**

**-XX:NewRatio** : NewRatio controls the size of young and **old** generation.

- **4.3) Permanent Generation (VM PARAMETERS for Permanent Generation)**

**-XX:PermSize**: It's initial value of Permanent Space which is allocated at startup of JVM.

**-XX:MaxPermSize**: It's maximum value of Permanent Space that JVM can allot up to.

**-XX:PermSize**: It's initial value of Permanent Space which is allocated at startup of JVM.

**-XX:MaxPermSize**: It's maximum value of Permanent Space that JVM can allot up to.

- **4.4) Other important VM (JVM) parameters for java heap in java >**

**-Xss** > Use this VM option to **adjust the maximum thread stack size**.

**-XX:MinHeapFreeRatio and -XX:MaxHeapFreeRatio**

JVM can grows or shrinks the heap to keep the proportion of free space to live objects within a specific range.

**-XX:+AggressiveHeap** is used for Garbage Collection Tuning setting.

- **5) Different Garbage collectors in detail >**

- **5.1) Serial collector / Serial GC (Garbage collector) in java**

- **5.1.1. Features of Serial GC (Garbage collector) in java >**

Serial GC (Garbage collector) is designed for the single threaded environments in java.

In Serial GC (Garbage collector) , both **minor and major garbage collections are done serially by one thread** (using a single virtual CPU) in java.

- **5.1.2. When to Use the Serial GC (garbage Collector) in java >**

applications that do not have low pause time requirements

- **5.1.3. Vm (JVM) option for enabling serial GC (garbage Collector) in java >**

**-XX:+UseSerialGC**

- **5.2) Throughput GC (Garbage collector) or Parallel collector in java**

- **5.2.1. Features of Throughput GC (Garbage collector) in java >**

Throughput garbage collector uses multiple threads to execute a minor collection and so reduces the serial execution time of the application in java.

- **5.2.2. When to Use the Throughput GC (Garbage collector) in java >**

The Throughput collector should be used when application can **afford low pauses** in java.

- **5.2.3. Vm (JVM) option for enabling throughput GC (Garbage collector) in java >**

**-XX:+UseParallelGC**

- **5.2.4. Goals for Throughput GC (Garbage collector) in java >**

Maximum pause time goal (Highest priority)

Throughput goal

Minimum footprint goal (Lowest priority)

- **5.2.5. Read in more detail about following features of Throughput GC (Garbage collector) in java**

- **5.3) Incremental low pause garbage collector (train low pause garbage collector) in java :**

Not used these days, was used in java 4.

- **5.4) Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java**

- **5.4.1. Features of Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java >**

Concurrent Mark Sweep (CMS) collector **collects the old/tenured generation** in java. Concurrent Mark Sweep (CMS) Collector **minimize the pauses** by doing most of the **garbage collection work concurrently with the application threads** in java.

- **5.4.2. When to Use the Concurrent Low Pause Collector in java**

Concurrent Low Pause Collector should be used if your **applications that require low garbage collection pause times** in java.

- **5.4.3. Vm (JVM) option for enabling Concurrent Mark Sweep (CMS) Collector in java >**

**-XX:+UseConcMarkSweepGC**

- **5.4.4. Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector working in detail in java >**

- **5.4.4.1 Major gc(garbage collection) in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector in java >**

**For each major collection** the CMS collector will **pause all the application threads for a brief period** at the **beginning** of the collection and toward the **middle** of the collection.

- **5.4.4.2 Minor gc (garbage collection) in Concurrent Mark Sweep (CMS) Collector / concurrent low pause collector >**

The minor collections is done in a manner **similar to the serial collector** although **multiple threads are used** to do the collection in java.

- **5.4.5. Heap Structure for CMS garbage Collector**

CMS garbage collectors didies heap into three sections: **young** generation, **old** generation, and **permanent** generation of a fixed memory size.

**Young** Generation is further divided into **Eden**, **S0 (Survivor space 0)** and **S1 (Survivor space 1)**.

- **5.4.6. Steps (in short) in GC (garbage collection) cycle in Concurrent Mark Sweep (CMS) Collector / concurrent low pause garbage collector in java >**

**Young GC (Generation garbage) Collection happens. Then, Than Old Generation GC (garbage Collection) happens.**

1. **initial mark > stop** all application threads; mark all live **objects**; **resume** all application threads
2. **concurrent mark >** do the concurrent **mark** (one processor is used for concurrent work)
3. **Remark > stop** all application threads; do the **remark**; **resume** all application threads
4. **sweep >** do the concurrent **sweep**, memory is freed up (one processor is used for concurrent work)
5. **reset >** do the concurrent **reset** (one processor is used for concurrent work)

- **5.5) G1 Garbage Collector (or Garbage First) in java**

- **5.5.1. The G1 garbage collector features -**

**G1 garbage collector** was **introduced in Java 7**

G1 garbage collector was designed to replace CMS garbage Collector.

G1 garbage collector is **parallel** and **concurrent**, and

**G1 Garbage Collector (or Garbage First) limits GC pause times and maximizes throughput.**

- **5.5.2. Vm** (JVM) option for enabling G1 Garbage Collector (or Garbage First) in java >

**-XX:+UseG1GC**

- **5.5.3. G1(Garbage First) collector functioning >**

CMS garbage collectors divides heap into three sections: young generation, old generation, and permanent generation of a fixed memory size.

The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions), **but** there is not a **fixed size** for them. This provides **greater flexibility in memory usage**.

- **5.5.4. When to use G1 garbage collector >**

G1 must be used when applications that require **large heaps** with limited GC latency.

- **5.5.5. When to switch from CMS (or old garbage collectors) to G1 garbage collector >**

**Full GC** durations are too **long** or too **frequent**.

- **5.5.6. The G1(Garbage First) collector working Step by Step >**

The G1 collector takes a different approach than CMS garbage collector in partitioning java heap memory.

- **5.5.6.1. G1(Garbage First) garbage collector Heap Structure >**

The heap is split/**partitioned** into **many fixed sized regions** (eden, survivor, old generation regions).

- **5.5.6.2. G1(Garbage First) garbage collector Heap Allocation >**

**Live objects** are **moved or copied** from **one region to another**.

As mentioned above there are following region in heap >

**Eden, survivor** and **old** generation region.

Also, **Humongous and unused** regions are there in heap.

- **5.5.6.3. Young Generation in G1 garbage collector**

Generally heap is divided into **2000 regions** by JVM.

**Minimum** size of region can be **1Mb** and

**Maximum** size of region can be **32Mb**.

Young GC in G1 garbage collector

- **Live objects** are copied or moved **to survivor regions**.
- If objects aging threshold is met it get promoted to **old** generation regions.

- It is **STW** (stop the world) event. Eden size and survivor size is calculated for the next young GC.

End of a Young GC with G1 garbage collector

At this stage **Live objects have been evacuated (copied or moved) to survivor regions or old generation regions.**

- 5.5.6.4. Old Generation Collection with G1 garbage collector

Initial Mark -

- It is **STW** (stop the world) event.
- Mark survivor regions (root regions) which may have references to objects in old generation.

Root Region Scanning -

- **Scan survivor regions for references into the old generation.**
- This happens while the **application continues to run.**

Concurrent Marking -

- **Find live objects over the entire heap.**
- This happens while the **application is running.**

Remark (Stop the World Event) -

- **Completes the marking of live object in the heap.**

Cleanup (Stop the World Event and Concurrent) -

- **Performs accounting on live objects and completely free regions.** (Stop the world)
- Young generation and old generation are reclaimed at the same time
- **Reset the empty regions** and return them to the free list. (Concurrent)

- **5.6) Difference between Serial GC (Garbage collector) vs Throughput GC (Garbage collector) in java?**

Serial collector **uses one thread to execute garbage collection.**

Throughput collector **uses multiple threads to execute garbage collection.**

- **6) What is Automatic Garbage Collection in JVM heap memory in java?**
- How to **Identify objects which are in use** in JVM heap memory in java?

**Objects** in use are those objects which are still needed by java program.

- **Which objects are not in use** in JVM heap memory in java?

**Objects** in use are those objects which are NOT still needed by java program.

- **7) Now let's understand how garbage collection is done using Marking and deletion in java.**

- **7.1) Step 1 > Marking**

Marking is a process in which gc (garbage collector) identifies which parts of memory (occupied by objects) are in use and which are not.

- **7.2) Step 2 > Deletion**

**Normal deletion removes all the unreferenced objects during process of garbage collection in java.**

## Miscellaneous Topics

### 10. What is significance of final in java?

1. **Final memberVariable/instanceVariable** of class must be initialized at time of declaration, once initialized final memberVariable cannot be assigned a new value.

2. **Final method** cannot be overridden, any attempt to do so will cause **compilation error**.

```

5 class Superclass {
6 final void method(){}
7 }
8
9 class Subclass extends Superclass{
10 void method(){}
11 }
```

Runtime polymorphism is not applicable on final methods because they cannot be overridden.

3. **Final class** cannot be extended, any attempt to do so will cause **compilation error**.

```

5 final class Superclass {
6 final void method(){}
7 }
8 class Subclass extends Superclass{} //final class cannot be extended
9 }
```

### 11. What is difference between using instanceof operator and getClass() in equals method?

If we use **instanceOf** it will return true for comparing current class with its subclass as well, but **getClass()** will return true only if exactly same class is compared. Comparison with any subclass will return false.

### 12. What is Immutable class?

**Any change made to object of immutable class produces new object.**

Example- String is Immutable class in java, any changes made to String object produces new String object.

### ***Creating Immutable class >***

- 1) **Final class** - Make class final so that it cannot be inherited
- 2) **private member variable** -> Making member variables private ensures that fields cannot be accessed outside class.
- 3) **final member variable** -> Make member variables final so that they can be assigned only once.
- 4) **Constructor** -> Initialize all fields in constructor.  
assign all mutable member variable using new keyword.
- 5) **Don't provide setter methods** in class/ provide only getter methods.
- 6) **object of immutable class** - Any change made to object of immutable class produces new object.  
**object of mutable class** - Any change made to object of mutable class doesn't produces new object.
  - **Integer, String are immutable class,**  
any changes made to object of these classes produces new object.  
so return reference variable of Integer.
  - **HashMap is mutable class,**  
any changes made to HashMap object won't produce new HashMap object.  
so return copy/clone of object, not reference variable of HashMap.\*/

## **13. Java Garbage Collection**

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### **Advantage of Garbage Collection**

- o It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.

- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## How can an object be unreferenced?

There are many ways:

- By nulling the reference
- By assigning a reference to another
- By anonymous object etc.

### 1) By nulling a reference:

1. Employee e=**new** Employee();

2. e=**null**;

### 2) By assigning a reference to another:

1. Employee e1=**new** Employee();

2. Employee e2=**new** Employee();

3. e1=e2;*//now the first object referred by e1 is available for garbage collection*

**n**

### 3) By anonymous object:

1. **new** Employee();

## finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

### 1. **protected void** finalize(){}

**Note:** The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

## gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

### 1. **public static void** gc(){}

**Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.**

### Simple Example of garbage collection in java

```

1. public class TestGarbage1{
2. public void finalize(){System.out.println("object is garbage collected");}
3. public static void main(String args[]){
4. TestGarbage1 s1=new TestGarbage1();
5. TestGarbage1 s2=new TestGarbage1();
6. s1=null;
7. s2=null;
8. System.gc();
9. }
10. }
```

#### **Test it Now**

```

object is garbage collected
object is garbage collected
```

### 14. Ways to create an object of a class?

There are five total ways to create objects in Java, which are explained below with their examples followed by bytecode of the line which is creating the object.

|                                                                 |                             |
|-----------------------------------------------------------------|-----------------------------|
| Using new keyword                                               | } → constructor gets called |
| Using <a href="#">newInstance()</a> method of Class class       | } → constructor gets called |
| Using <a href="#">newInstance()</a> method of Constructor class | } → constructor gets called |
| Using clone() method                                            | } → no constructor call     |
| Using deserialization                                           | } → no constructor call     |

If you will execute program given in end, you will see method 1, 2, 3 uses the constructor to create the object while 4, 5 doesn't call the constructor to create the object.

## 1. Using new keywords

It is the most common and regular way to create an object and a very simple one also. By using this method we can call whichever constructor we want to call (no-arg constructor as well as parameterized).

```
Employee emp1 = new Employee();
0: new #19 // class org/programming/mitra/exercises/Emp
yee
3: dup
4: invokespecial #21 // Method org/programming/mitra/exercises/Empl
oyee."":()V
```

## 2. Using newInstance() method of Class class

We can also use the newInstance() method of a Class class to create an object. This newInstance() method calls the no-arg constructor to create the object.

We can create an object by newInstance() in the following way:

```
Employee emp2 = (Employee) Class.forName("org.programming.mitra.exercises.Emp
loyee").newInstance();
```

Or

```
Employee emp2 = Employee.class.newInstance();
51: invokevirtual #70 // Method java/lang/Class.newInstance:()Ljava/lan
g/Object;
```

## 4. Using newInstance() method of Constructor class

Similar to the newInstance() method of Class class, There is one newInstance() method in the java.lang.reflect.Constructor class which we can use to create objects. We can also call parameterized constructor, and private constructor by using this newInstance() method.

```
Constructor<Employee> constructor = Employee.class.getConstructor();
Employee emp3 = constructor.newInstance();
111: invokevirtual #80 // Method java/lang/reflect/Constructor.newInstance:
([Ljava/lang/Object;)Ljava/lang/Object;
```

Both newInstance() methods are known as reflective ways to create objects. In fact newInstance() method of Class class internally uses newInstance() method of Constructor class. That's why the later one is preferred and also used by different frameworks like Spring, Hibernate,

Struts etc. To know differences between both newInstance() methods read [Creating objects through Reflection in Java with Example](#).

#### 4. Using clone() method:

Whenever we call clone() on any object, the JVM actually creates a new object for us and copies all content of the previous object into it. Creating an object using the clone method does not invoke any constructor.

To use clone() method on an object we need to implement Cloneable and define the clone() method in it.

```
Employee emp4 = (Employee) emp3.clone();
162: invokevirtual #87 // Method org/programming/mitra/exercises/Employee.cl
one ()Ljava/lang/Object;
```

Java cloning is the most debatable topic in Java community and it surely does have its drawbacks but it is still the most popular and easy way of creating a copy of any object until that object is full filling mandatory conditions of Java cloning. I have covered cloning in details in a 3 article long [Java Cloning Series](#) which includes ([Java Cloning And Types Of Cloning \(Shallow And Deep\) In Details With Example](#), [Java Cloning - Copy Constructor Versus Cloning](#), [Java Cloning - Even Copy Constructors Are Not Sufficient](#)), go ahead and read them if you want to know more about cloning.

#### 5. Using deserialization:

Whenever we serialize and deserialize an object, the JVM creates a separate object for us. In deserialization, the JVM doesn't use any constructor to create the object.

To deserialize an object we need to implement a Serializable interface in our class.

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data.obj"))
;
Employee emp5 = (Employee) in.readObject();
261: invokevirtual #118 // Method java/io/ObjectInputStream.readObject:()L
java/lang/Object;
```

As we can see in the above bytecode snippets, all 4 methods are called and get converted to invokevirtual (object creation is directly handled by

these methods) except the first one, which got converted to two calls: one is new and other is invokespecial (call to constructor).

## Example

Let's consider an Employee class for which we are going to create the objects:

```
class Employee implements Cloneable, Serializable {
 private static final long serialVersionUID = 1L;
 private String name;
 public Employee() {
 System.out.println("Employee Constructor Called...");
 }
 public String getName() {
 return name;
 }
 public void setName(String name) {
 this.name = name;
 }
 @Override
 public int hashCode() {
 final int prime = 31;
 int result = 1;
 result = prime * result + ((name == null) ? 0 : name.hashCode());
 return result;
 }
 @Override
 public boolean equals(Object obj) {
 if (this == obj)
 return true;
 if (obj == null)
 return false;
 if (getClass() != obj.getClass())
 return false;
 Employee other = (Employee) obj;
 if (name == null) {
 if (other.name != null)
 return false;
 } else if (!name.equals(other.name))
 return false;
 return true;
 }
 @Override
 public String toString() {
 return "Employee [name=" + name + "]";
 }
 @Override
 public Object clone() {
 Object obj = null;
 try {
 obj = super.clone();
 } catch (CloneNotSupportedException e) {
 e.printStackTrace();
 }
 }
}
```

```

 }
 return obj;
 }
}

```

In the below Java program we are going to create Employee objects in all 5 ways. You can also find the source code at [GitHub](#).

```

public class ObjectCreation {
 public static void main(String... args) throws Exception {
 // By using new keyword
 Employee emp1 = new Employee();
 emp1.setName("Naresh");
 System.out.println(emp1 + ", hashCode : " + emp1.hashCode());
 // By using Class class's newInstance() method
 Employee emp2 = (Employee) Class.forName("org.programming.mitra.exercises.Employee")
 .newInstance();
 // Or we can simply do this
 // Employee emp2 = Employee.class.newInstance();
 emp2.setName("Rishi");
 System.out.println(emp2 + ", hashCode : " + emp2.hashCode());
 // By using Constructor class's newInstance() method
 Constructor<Employee> constructor = Employee.class.getConstructor();
 Employee emp3 = constructor.newInstance();
 emp3.setName("Yogesh");
 System.out.println(emp3 + ", hashCode : " + emp3.hashCode());
 // By using clone() method
 Employee emp4 = (Employee) emp3.clone();
 emp4.setName("Atul");
 System.out.println(emp4 + ", hashCode : " + emp4.hashCode());
 // By using Deserialization
 // Serialization
 ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(
 "data.obj"));
 out.writeObject(emp4);
 out.close();
 //Deserialization
 ObjectInputStream in = new ObjectInputStream(new FileInputStream("dat
a.obj"));
 Employee emp5 = (Employee) in.readObject();
 in.close();
 emp5.setName("Akash");
 System.out.println(emp5 + ", hashCode : " + emp5.hashCode());
 }
}

```

This program will give the following output:

```

Employee Constructor Called...
Employee [name=Naresh], hashCode : -1968815046
Employee Constructor Called...
Employee [name=Rishi], hashCode : 78970652
Employee Constructor Called...
Employee [name=Yogesh], hashCode : -1641292792
Employee [name=Atul], hashCode : 2051657
Employee [name=Akash], hashCode : 63313419

```

## 15. What are the types of references in Java?

In Java there are four types of references differentiated on the way by which they are garbage collected.

1. Strong References
2. Weak References
3. Soft References
4. Phantom References

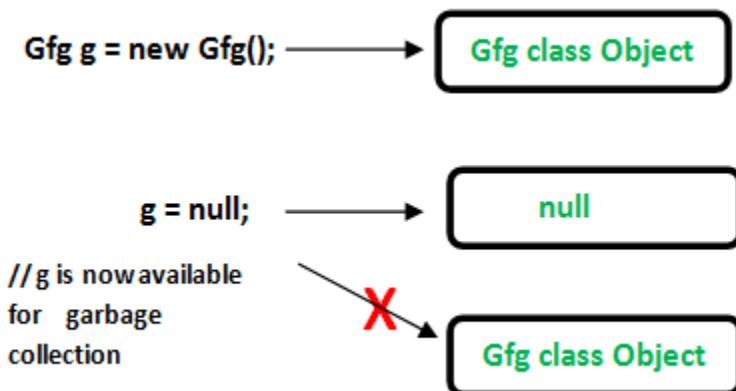
## 16. What are ‘Strong References’?

This is the default type/class of Reference Object. Any object which has an active strong reference are not eligible for garbage collection. The object is garbage collected only when the variable which was strongly referenced points to null.

```
MyClass obj = new MyClass();
```

Here ‘obj’ object is strong reference to newly created instance of MyClass, currently obj is active object so can’t be garbage collected.

```
obj = null;
// 'obj' object is no longer referencing to the instance.
So the 'MyClass' type object is now available for garbage
collection.
```



```
// Java program to illustrate Strong reference
class Gfg
{
 //Code..
}
public class Example
{
```

```

public static void main(String[] args)
{
 //Strong Reference - by default
 Gfg g = new Gfg();

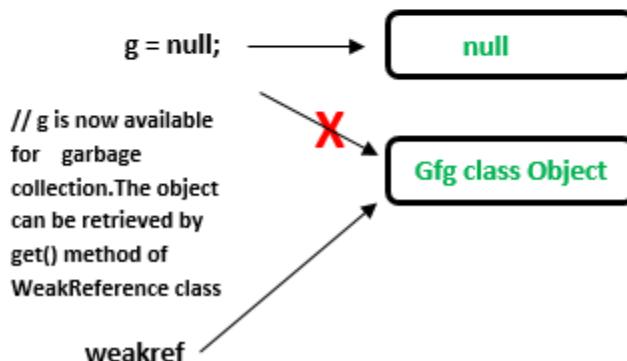
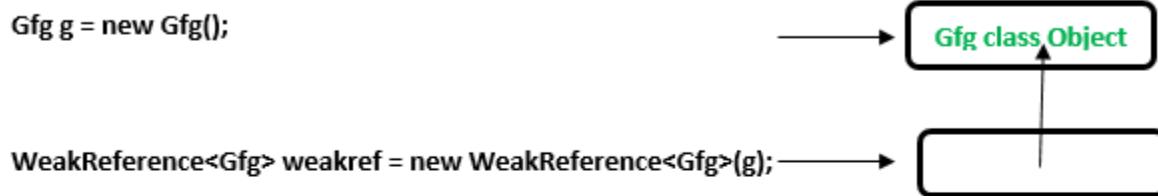
 //Now, object to which 'g' was pointing earlier is
 //eligible for garbage collection.
 g = null;
}
}

```

## 17. What are ‘Weak References’?

Weak Reference Objects are not the default type/class of Reference Object and they should be explicitly specified while using them.

- This type of reference is used in WeakHashMap to reference the entry objects .
- If JVM detects an object with only weak references (i.e. no strong or soft references linked to any object object), this object will be marked for garbage collection.
- To create such references [java.lang.ref.WeakReference](#) class is used.
- These references are used in real time applications while establishing a DBConnection which might be cleaned up by Garbage Collector when the application using the database gets closed.



`//Java Code to illustrate Weak reference`

```
import java.lang.ref.WeakReference;

class Gfg

{

 //code

 public void x()

 {

 System.out.println("GeeksforGeeks");

 }

}

public class Example

{

 public static void main(String[] args)

 {

 // Strong Reference

 Gfg g = new Gfg();

 g.x();

 // Creating Weak Reference to Gfg-type object to which

 'g'

 // is also pointing.

 WeakReference<Gfg> weakref = new WeakReference<Gfg>(g);

 //Now, Gfg-type object to which 'g' was pointing earlier

 //is available for garbage collection.

 //But, it will be garbage collected only when JVM needs

 memory.

 g = null;
```

```
// You can retrieve back the object which
// has been weakly referenced.
// It successfully calls the method.
g = weakref.get();

g.x();
}
}
```

Output:

```
GeeksforGeeks
GeeksforGeeks
```

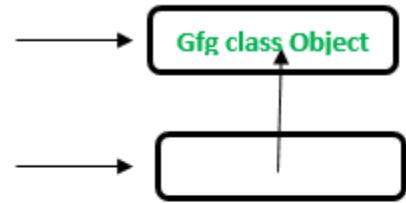
Two different levels of weakness can be enlisted: **Soft and Phantom**

### **18. What are ‘Soft References’?**

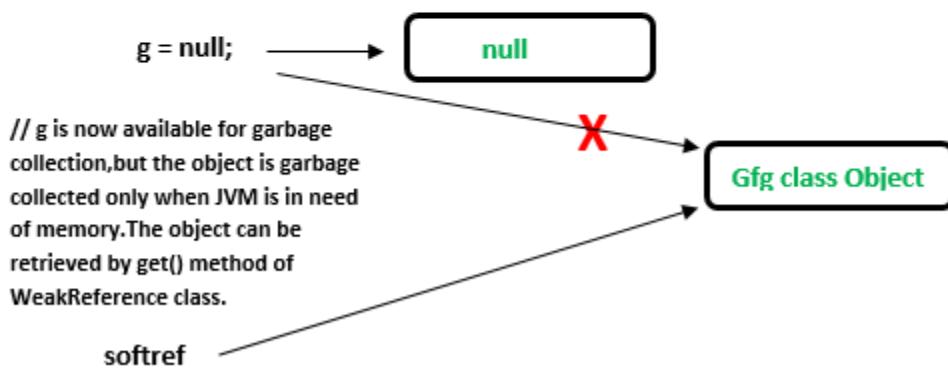
In Soft reference, even if the object is free for garbage collection then also its not garbage collected, until JVM is in need of memory badly. The objects gets cleared from the memory when JVM runs out of memory. To create such references [java.lang.ref.SoftReference](#) class is

used.

```
Gfg g = new Gfg();
```



```
SoftReference<Gfg> softref = new SoftReference<Gfg>(g);
```



```

//Code to illustrate Soft reference
import java.lang.ref.SoftReference;
class Gfg
{
 //code..
 public void x()
 {
 System.out.println("GeeksforGeeks");
 }
}

public class Example
{

```

```

public static void main(String[] args)
{
 // Strong Reference
 Gfg g = new Gfg();
 g.x();

 // Creating Soft Reference to Gfg-type object to which
 'g'
 // is also pointing.
 SoftReference<Gfg> softref = new SoftReference<Gfg>(g);

 // Now, Gfg-type object to which 'g' was pointing
 // earlier is available for garbage collection.
 g = null;

 // You can retrieve back the object which
 // has been weakly referenced.
 // It successfully calls the method.
 g = softref.get();

 g.x();
}
}

```

**Output:**

```

GeeksforGeeks
GeeksforGeeks

```

### 19. What are ‘Phantom/Ghost References’?

The objects which are being referenced by phantom references are eligible for garbage collection. But, before removing them from the memory, JVM puts them in a queue called

'reference queue'. They are put in a reference queue after calling finalize() method on them. To create such references [java.lang.ref.PhantomReference](#) class is used.

```
//Code to illustrate Phantom reference

import java.lang.ref.*;
class Gfg
{
 //code
 public void x()
 {
 System.out.println("GeeksforGeeks");
 }
}

public class Example
{
 public static void main(String[] args)
 {
 //Strong Reference
 Gfg g = new Gfg();
 g.x();

 //Creating reference queue
 ReferenceQueue<Gfg> refQueue = new ReferenceQueue<Gfg>();

 //Creating Phantom Reference to Gfg-type object to which
 'g'
 //is also pointing.
 PhantomReference<Gfg> phantomRef = null;
```

```

phantomRef = new PhantomReference<Gfg>(g, refQueue);

//Now, Gfg-type object to which 'g' was pointing
//earlier is available for garbage collection.
//But, this object is kept in 'refQueue' before
//removing it from the memory.

g = null;

//It always returns null.

g = phantomRef.get();

//It shows NullPointerException.

g.x();

}

}

```

**Runtime Error:**

```

Exception in thread "main" java.lang.NullPointerException
at Example.main(Example.java:31)

```

**Output:**

GeeksforGeeks

## 20. Difference between FileReader and BufferedReader in java file IO

|   | <b>BufferedReader</b>                      | <b>FileReader</b>                          |
|---|--------------------------------------------|--------------------------------------------|
| 1 | <b>BufferedReader</b> is <b>buffered</b> . | <b>FileReader</b> is <b>not buffered</b> . |

|   |                                                                                                                                                                                                                                                                                                                                                                            |                                                                                                                                                                                                                               |
|---|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 2 | <b>BufferedReader reads characters from another Reader (Eg - FileReader)</b>                                                                                                                                                                                                                                                                                               | <b>FileReader reads characters from a file.</b>                                                                                                                                                                               |
| 3 | <p>when <b>BufferedReader.read()</b> is called mostly data is read from the buffer.</p> <p>When data is not available available in buffer a call is made to read system file and lot of characters are kept in buffer.</p>                                                                                                                                                 | <p>Every time <b>FileReader.read()</b> is called a call is made to read a system file.</p> <p><b>FileReader.read()</b> reads <b>2 byte</b> (16-bit) at a time.</p>                                                            |
| 4 | <p>A <b>BufferedReader</b> enables another Reader to <b>buffer the characters and supports the mark and reset methods.</b></p> <p>An internal buffer array is created when the BufferedReader is created.</p> <p>As characters from the Reader are read or skipped, the internal buffer is refilled as necessary from the contained Reader, many characters at a time.</p> | <p>A <b>FileReader</b> obtains characters from a file in a file system.</p> <p>And does <b>not supports mark and reset methods.</b></p>                                                                                       |
| 5 | BufferedReader is much <b>faster</b> as compared to FileReader.                                                                                                                                                                                                                                                                                                            | FileReader is <b>slower</b> as compared to BufferedReader.                                                                                                                                                                    |
| 6 | <p>Example -</p> <p>As we discussed above that when BufferedReader.read() is called mostly data is read from the buffer.</p> <p><b>A BufferedReader reads from FileReader, will request lot of data from the FileReader (128 characters or so... not exact figure). Thus only 2 calls will be made for reading 256 characters from file.</b></p>                           | <p>Example -</p> <p>As we discussed in point above that every time FileReader.read() is called a call is made to read a system file.</p> <p><b>A FileReader will make 256 calls for reading 256 characters from file.</b></p> |

|                                                                                                                                                                                                                         |                                                                            |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|
| <p>Another Example - <b>Real world Example</b> - You must have seen youtube videos where video is buffered before you actually start watching it, <b>buffering</b> overall improves your video watching experience.</p> | <p>No buffering will make your videos watching experience a nightmare.</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------|

## 21. Why do we assign a parent reference to the child object in Java?

First, a clarification of terminology: we are assigning a **Child** object to a variable of type **Parent**. **Parent** is a reference to an object that happens to be a subtype of **Parent**, a **Child**.

It is only useful in a more complicated example. Imagine you add `getEmployeeDetails` to the class **Parent**:

```
public String getEmployeeDetails() {
 return "Name: " + name;
}
```

We could override that method in **Child** to provide more details:

```
@Override
public String getEmployeeDetails() {
 return "Name: " + name + " Salary: " + salary;
}
```

Now you can write one line of code that gets whatever details are available, whether the object is a **Parent** or **Child**:

```
parent.getEmployeeDetails();
```

The following code:

```
Parent parent = new Parent();
parent.name = 1;
Child child = new Child();
child.name = 2;
```

```

child.salary = 2000;

Parent[] employees = new Parent[] { parent, child };

for (Parent employee : employees) {
 employee.getEmployeeDetails();
}

```

Will result in the output:

Name: 1

Name: 2 Salary: 2000

We used a **Child** as a **Parent**. It had specialized behavior unique to the **Child** class, but when we called `getEmployeeDetails()` we could ignore the difference and focus on how **Parent** and **Child** are similar. This is called [subtype polymorphism](#).

Your updated question asks why `Child.salary` is not accessible when the `Child` object is stored in a `Parent` reference. The answer is the intersection of "polymorphism" and "static typing". Because Java is statically typed at compile time you get certain guarantees from the compiler but you are forced to follow rules in exchange or the code won't compile. Here, the relevant guarantee is that every instance of a subtype (e.g. `Child`) can be used as an instance of its supertype (e.g. `Parent`). For instance, you are guaranteed that when you access `employee.getEmployeeDetails()` or `employee.name` the method or field is defined on any non-null object that could be assigned to a variable `employee` of type `Parent`. To make this guarantee, the compiler considers only that static type (basically, the type of the variable reference, `Parent`) when deciding what you can access. So you cannot access any members that are defined on the runtime type of the object, `Child`.

When you truly want to use a `Child` as a `Parent` this is an easy restriction to live with and your code will be usable for `Parent` and all its subtypes. When that is not acceptable, make the type of the reference `Child`.

# Java 8

## 1. Functional Interface

### 1. What are Functional interface in java 8?

Functional interface are those interfaces that can exactly have one abstract method in java

### 2. How can we create/make interface a FunctionalInterface in java 8?

We can make interface a FunctionalInterface by using **annotation @FunctionalInterface** in java 8.

### 3. Advantage of making Functional interface in java 8?

It is very handy in ensuring that your interface has exactly defined one abstract method in java 8.

Functional interface can be used with Lambda expression, which makes code much neat, clean and easy to read

### 4. Program/Example of defining/creating @FunctionalInterface and using it in java 8 OR Full Program/Example of defining/creating @FunctionalInterface - And write Anonymous inner class to implement method of FunctionalInterface in java 8 >

```
//Define/Create @FunctionalInterface in java 8
@FunctionalInterface
interface MyInterface {
 public abstract void myMethod();
}
public class MainClass{
 public static void main(String...args) {

 //Write Anonymous inner class to implement method of MyInterface
 (FunctionalInterface)
 MyInterface myInterface = new MyInterface() {
 @Override
 public void myMethod() {
 System.out.println("xx");
 }
 };

 //Call myMethod()
 myInterface.myMethod();
 }
}
/* OUTPUT
xx
*/
```

So. in this program we **created @FunctionalInterface**  
 And then in main method We wrote **Anonymous inner class to implement MyInterface (FunctionalInterface)**

5. Full Program/Example of defining/creating @FunctionalInterface - And write Lambda expression to implement method of FunctionalInterface in java 8 >

```
//Define/Create @FunctionalInterface in java 8
@FunctionalInterface
interface MyInterface {
 public abstract void myMethod();
}
public class MainClass{
 public static void main(String...args) {
```

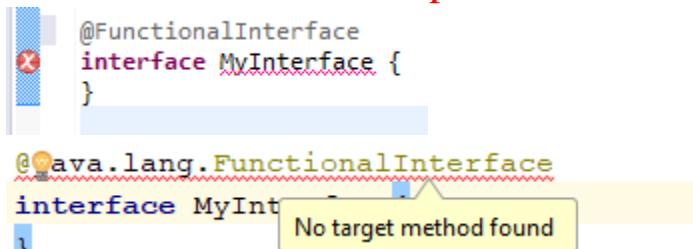
```

 //Write LAMBDA EXPRESSION to implement method of MyInterface
 (@FunctionalInterface)
 MyInterface myInterface = () -> {
 System.out.println("xx");
 };

 //Call myMethod()
 myInterface.myMethod();
}
/* OUTPUT
xx
*/

```

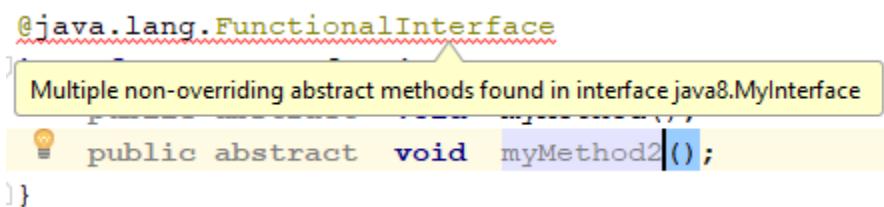
6. If no abstract method is defined in functional interface we will face **compilation error** in java 8



7. What's exact error in eclipse in java 8 ?

**"Invalid '@FunctionalInterface' annotation; MyInterface is not a functional interface"**

8. If more than one abstract method is defined in functional interface we will face **compilation error** in java 8



9. Can @FunctionalInterface can have static and default method in java 8 ?

Yes. Example of FunctionalInterface having static and default method in java 8 >

```
@FunctionalInterface
interface MyInterface1 {
 abstract void myMethod();
 default void defaultMethod() {}
 static void staticMethod() {}
}
```

## Default Methods

1. What is default method in interface in java 8?

Now, we can add add default(non-abstract) method in interfaces in java 8  
(Before java 8, we could only write abstract method in interfaces in java 8)

By making method as **default** in Interface we can make concrete (non-abstract) method in interface. [i.e. we can define method in interface]

default method are also called

- **defender** methods or
- **extension** methods in java 8.

default method are **public** by default.

2. Full Program/Example to create and use default method of interface in Java 8 >

```
interface Animals {
 /*
 * Define default method in java 8
 */
 default void food() {
 System.out.println("Animal eat food");
 }
}

class Lion implements Animals {
```

```
//No no need to override any method of interface (As there is no abstract method in
interface)
}

public class MainClass{
 public static void main(String...args) {
 Animals animals = new Lion();
 animals.food();
 }
}
/* OUTPUT
Animal eat food
*/
```

### 3. What is Advantage of default methods in java 8?

There is no need to override default methods of interface in implementing class.

Example - Lion class didn't override default method food.

Before java 8 - Every class implementing interface was needed to override all the methods of interface (As we could only define abstract methods in interface).

### 4. Can we override default method of interface in java 8?

Yes, we can override default method of interface in java 8.

```
interface Animals {
 /* Define default method in java 8 */
 default void food() {
 System.out.println("Animal eat food");
 }
}
class Lion implements Animals {
 /*If we want - we can override default method of interface in java
8*/
 @Override
 public void food() {
 System.out.println("Lion eat - flesh");
 }
}
public class MainClass{
 public static void main(String...args) {
 Animals animals = new Lion();
```

```

 animals.food();
 }
}
/* OUTPUT
Lion eat - flesh
*/

```

5. How to Override default methods in java 8 - in anonymous inner class >

```

interface Animals {
 /* Define default method in java 8 */
 default void food() {
 System.out.println("Animal eat food");
 }
}
public class MainClass{
 public static void main(String...args) {
 /*override default methods in java 8 - in anonymous inner
 class*/
 Animals lion = new Animals() {
 @Override
 public void food() {
 System.out.println("Lion eat - flesh");
 }
 };
 lion.food();
 }
}
/* OUTPUT
Lion eat - flesh
*/

```

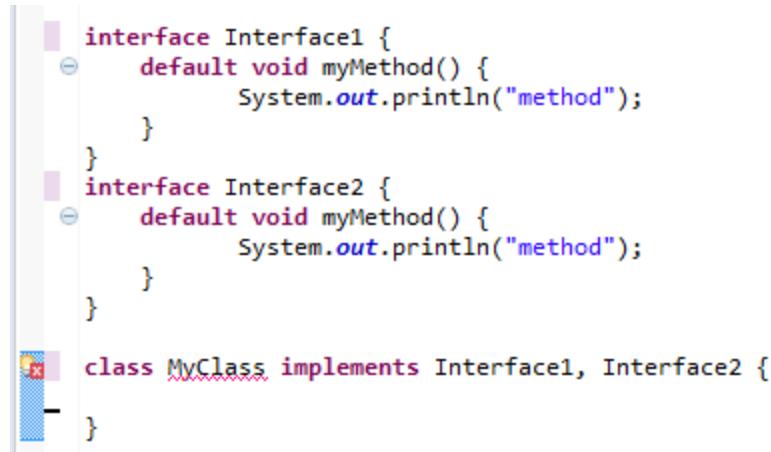
6. What if two interfaces have declared a default method with same name and a class tries to implement both the interface in java 8 ?

We will face **compilation error** that Interface1 and Interface2 have same name for default myMethod.

It is also called **DIAMOND** problem in java

## Exact error shown in eclipse >

"Duplicate default methods named method with the parameters () and () are inherited from the types Interface1 and Interface2"



```

interface Interface1 {
 default void myMethod() {
 System.out.println("method");
 }
}
interface Interface2 {
 default void myMethod() {
 System.out.println("method");
 }
}
class MyClass implements Interface1, Interface2 {
}

```

## Solution >

We need to override myMethod in MyClass to avoid this error in java 8.

```

interface Interface1 {
 default void myMethod() {
 System.out.println("method");
 }
}

interface Interface2 {
 default void myMethod() {
 System.out.println("method");
 }
}

class MyClass implements Interface1, Interface2 {
 public void myMethod() {

```

```

 System.out.println("myMethod");

 }

}

```

## Static Methods

### 1. static method in interface in java 8

Now, we can **write** static method in interface in java 8. (Before java 8, we could only write abstract method in interfaces in java 8)

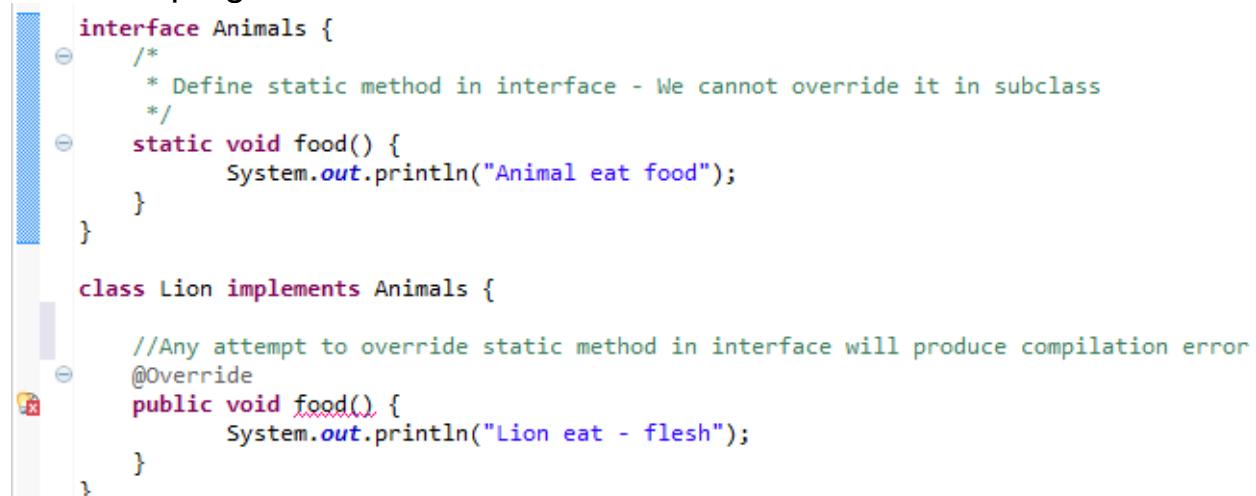
We can **define** these static methods in interface, just like normal static method. static method are **public** by default in java 8.

### 2. Can we override static method of interface in subclass in java 8?

We **cannot** override static method of interface in subclass in java 8.

**No**, we cannot override static method of interface in subclass in java 8.

Let's see program -



```

interface Animals {
 /*
 * Define static method in interface - We cannot override it in subclass
 */
 static void food() {
 System.out.println("Animal eat food");
 }
}

class Lion implements Animals {

 //Any attempt to override static method in interface will produce compilation error
 @Override
 public void food() {
 System.out.println("Lion eat - flesh");
 }
}

```

First, we defined static method in interface, we cannot override it in subclass.

Then, we saw **attempt to override static method of interface produced compilation error in java 8.**

3. Full Program/Example to create and use static method of interface in Java 8 >

```
/*
Full Example to use static method of interface in Java 8
*/
interface Animals {
 /*
 * Create/Define static method in interface
 */
 public static void food() {
 System.out.println("Animal eat food");
 }
}
public class MainClass{
 public static void main(String...args) {
 //Call static method of interface
 Animals.food();
 }
}

/* OUTPUT
Animal eat food
*/
```

4. Full Program/Example to create and use static method of interface in Java 8 >

```
/*
Full Example to use static method of interface in Java 8
*/
interface Animals {
 /*
 * Create/Define static method in interface
 */
 public static void food() {
 System.out.println("Animal eat food");
 }
}
public class MainClass{
 public static void main(String...args) {
 //Call static method of interface
 Animals.food();
 }
}
```

```

 }
}

/* OUTPUT
Animal eat food
*/

```

### 5. Advantage of static method in interface in java 8?

We can make **utility interface** which consists of static methods in java 8.  
For example -

[java.util.Collections](#) is a **utility class** which consists of static methods that operate on or return Collection in java.

## Lambda Expressions

### 1. What is lambda expressions?

- we can write lambda expression to replace the [anonymous inner class](#).
- lambda expression make code very neat and clean.
- lambda expression are very to read. So, they make code more readable.

### 2. How to use Lambda expression?

Example 1.1 > **Before Java 8 - Sort String using Using Local class - Without Lambda expression >**

```

import java.util.Arrays;
import java.util.Comparator;

public class SortStringArrayWithoutLambdaExpressionExample {

 public static void main(String... args) {
 String[] stringArray = {"ab", "ef", "cd"};
 //Create Local class
 }
}

```

```

class StringSort implements Comparator<String> {

 public int compare(String a, String b) {
 return a.compareTo(b);
 }
}

//Before Java 8 - Sort String using Using Local class - Without Lambda expression

System.out.println("Before Java 8 - Sort StringArray using Using "
+ " > Local class - i.e. Without Lambda expression");

Arrays.sort(stringArray, new StringSort());

//Display StringArray

for (String str : stringArray) {
 System.out.print(str + " ");
}

}
/*
Output
Before Java 8 - Sort StringArray using Using > Local class - i.e. Without Lambda expression
ab cd ef
*/

```

Example 1.2 > Before Java 8 - Sort StringArray using > **Anonymous Inner class** - i.e. Without Lambda expression >

```

Arrays.sort(stringArray, new Comparator<String>() {
 @Override
 public int compare(String a, String b) {

```

```

 return a.compareTo(b);
 }
});
```

### Example 1.3 > In Java 8 - Sort StringArray using > Lambda expression (Replace Anonymous Inner class with Lambda expression)

```

import java.util.Arrays;

public class SortStringArrayLambdaExpressionExample2 {
 public static void main(String... args) {
 String[] stringArray = { "ab", "ef", "cd" };
 System.out.println("In Java 8 - Sort StringArray using > Lambda expression");
 Arrays.sort(stringArray, (String a, String b) -> {
 return a.compareTo(b);
 });

 // Display StringArray
 for (String str : stringArray) {
 System.out.print(str + " ");
 }
 }
}
/* Output
In Java 8 - Sort StringArray using > Lambda expression
ab cd ef
*/
```

### Example 1.4 > You can shorten up the above lambda expression >

```
Arrays.sort(stringArray, (a, b) -> {
 return a.compareTo(b);
});
```

Example 1.5 > **If there is only** Single line in implementation, we can also remove curly braces (With single line you can also remove return statement) >

```
Arrays.sort(stringArray, (a, b) -> a.compareTo(b));
```

Example 1.6 > **In Java 8** - Sort [StringList](#) using > **Lambda expression (using only** Single line) >

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class SortStringListExpressionExample {
 public static void main(String... args) {
 //Declare StringArray
 String[] stringArray = { "ab", "ef", "cd" };

 //Convert String Array to String List
 List<String> stringList = Arrays.asList(stringArray);

 //In Java 8 - Sort StringList using > Lambda expression - in one line
 Collections.sort(stringList, (a, b) -> a.compareTo(b));

 //Display StringList
 System.out.println(stringList);
 }
}
```

```
/* Output
[ab, cd, ef]
*/
```

### 3. Sum of numbers

Example 2.1 > In Java 8 - Sum of two numbers program using > **Lambda expression**

```
@FunctionalInterface
interface CalculatorInterface<A> {
 public abstract A sumMethod(A val1, A val2);
}
public class LambdaExpression_FunctionalInterface_calculatorSum {
 public static void main(String[] args) {
 // Provide implementation (definition) of sumMethod - using Lambda
 // expression
 // A will be type Integer
 CalculatorInterface<Integer> sum = (Integer val1, Integer val2) ->
{
 return val1 + val2;
};

 // Call sumMethod
 Integer result = sum.sumMethod(2, 3);
 System.out.println(result); // 5
 }
}
/* OUTPUT
5
*/
```

Example 2.2 > You can shorten up the above lambda expression >

```
CalculatorInterface<Integer> sum = (val1, val2) -> {
 return val1 + val2;
};
```

Example 2.3 > **If there is only** Single line in implementation, we can also remove curly braces (With single line you can also remove return statement) >

```
CalculatorInterface<Integer> sum = (val1, val2) -> val1 + val2;
```

#### 4. To square a number

Example 3.1 > **In Java 8** - square of number program using > **Lambda expression**

```
@FunctionalInterface
interface CalculatorInterface<A> {
 A squareMethod(A val);
}

public class LambdaExpression_FunctionalInterface_calculatorSquare {
 public static void main(String[] args) {
 //Provide implementation (definition) of squareMethod using > Lambda
 //expression
 // A will be type Integer
 CalculatorInterface<Integer> square = (val) -> (val * val);
 // Call squareMethod
 Integer result = square.squareMethod(2);
 System.out.println(result); // 4
 }
}
/* Output
4
*/
```

Example 3.2 > **Before Java 8** - square of number program Using anonymousInnerClass - **Without Lambda expression** >

```
CalculatorInterface<Integer> square = new CalculatorInterface() {
 @Override
 public Object squareMethod(Object val) {
 return ((Integer)val * (Integer)val);
 }
};
```

## 5. How to use Lambda expression with threads?

Example 4.1 > Before Java 8 - Create thread, Implement Runnable interface using > **Anonymous Inner class** - i.e. Without Lambda expression

```
public class WithoutLambdaExpressionThreadExample {
 public static void main(String[] args) {
 System.out.println("1 - Create thread, Implement Runnable interface using
> Anonymous inner class");
 // Create thread, Implement Runnable interface using Anonymous inner
 class
 new Thread(new Runnable() {
 @Override
 public void run() {
 System.out.println("Thread-1");
 }
 }).start();
 }
}
/*OUTPUT
1 - Create thread, Implement Runnable interface using > Anonymous inner class
Thread-1
*/
```

Example 4.2 > In Java 8 - Create thread, Implement Runnable interface using > **Lambda expression**

```
public class LambdaExpressionThreadExample2 {
 public static void main(String[] args) {
 System.out.println("1 - Implement Runnable interface using > Lambda
expression");
 // Implement Runnable interface using > Lambda expression
 new Thread(() -> {
 System.out.println("Thread-1");
 }).start();
 }
}
/*OUTPUT
1 - Implement Runnable interface using > Lambda expression
Thread-1
*/
```

Example > **If there is only** Single line in implementation, we can also remove curly braces (With single line you can also remove return statement) >

```
new Thread(() -> System.out.println("Thread1")).start();
```

## 6. How to use Lambda expression - String to Integer conversion?

Example > **In Java 8** - convert String To Integer using Lambda expression

```
/*
convert String To Integer using Lambda expression
*/
@FunctionalInterface
interface MyInterface<A, B> {
 A convertStringToIntegerMethod(B stringVal);
}

public class LambdaExpressionExample {
 public static void main(String[] args) {
 // Provide implementation of convertStringToInteger using Lambda expression
 // A will be type Integer
 // B will be of type String
 MyInterface<Integer, String> integerVal= (stringVal) ->
 Integer.valueOf(stringVal);
 // Call convertStringToInteger
 Integer result = integerVal.convertStringToIntegerMethod("12");
 System.out.println("Integer = "+result); // 12
 }
}
/* OUTPUT
Integer = 12
*/
```

## 7. Final variable can be accessed in lambda expression in java 8 (Behavior same as anonymous inner class)

```

@FunctionalInterface
interface CalculatorInterface {
 Integer sumMethod(Integer val1, Integer val2);
}

public class LambdaExpressionScopeExample {
 public static void main(String[] args) {
 // Final Local variable can be accessed in lambda expression in java 8
 final int x = 1; // Final Local variable

 // Provide implementation (definition) of sumMethod - using Lambda
 // expression

 CalculatorInterface sum = (val1, val2) -> {
 return val1 + val2 + x;
 };

 // Call sumMethod
 Integer result = sum.sumMethod(2, 3);
 System.out.println("sumResult = "+result); // 6
 }
}
/*
sumResult = 6
*/

```

8. **non-Final** variable can be **accessed** in lambda expression in java 8

Though **non-Final** variable can be **accessed** in lambda expression, **But it is effectively final in lambda expression** in java 8 ( Again Behavior same as anonymous inner class).

So, you use non-final variable in Lambda expression BUT you **cannot modify x in lambda expression.**

Any attempt to modify x will produce compilation error.

What will be **exact compilation error** in eclipse in java 8 ?

**"Local variable x defined in an enclosing scope must be final or effectively final."**

```
@FunctionalInterface
interface CalculatorInterface {
 Integer sumMethod(Integer val1, Integer val2);
}

public class LambdaExpressionScopeExample {
 public static void main(String[] args) {
 // Though non-Final local variable can be accessed in lambda expression
 // But it is effectively final in lambda expression.
 // So, you cannot modify x in lambda expression.
 // Any attempt to modify x will produce compilation error in java 8.

 int x = 1; // Non-Final Local variable

 CalculatorInterface sum = (val1, val2) -> {
 x = x + 1; //COMPILATION ERROR
 return val1 + val2 + x;
 };
 }
}
```

## 9. Accessing Instance and static variable in Lambda Expression in java 8 >

1. Instance variable **can be accessed (and modified unlike local variable)** in lambda expression in java 8
2. Static variable **can be accessed (and modified unlike local variable)** in lambda expression in java 8

```

@FunctionalInterface
interface CalculatorInterface {
 Integer sumMethod(Integer val1, Integer val2);
}

public class LambdaExpressionScopeExample {
 // Instance variable can be accessed in lambda expression in java 8
 int instanceVariable = 1; // Instance variable

 // Static variable can be accessed in lambda expression in java 8
 static int staticVariable = 2; // Static variable

 public static void main(String[] args) {

 LambdaExpressionScopeExample obj = new LambdaExpressionScopeExample();
 // Provide implementation (definition) of sumMethod - using Lambda
 expression
 CalculatorInterface sum = (val1, val2) -> {
 return val1 + val2 +
 + obj.instanceVariable // Access Instance variable in
 lambda expression
 + staticVariable; // Access static variable in lambda
 expression
 };
 // Call sumMethod
 Integer sumResult = sum.sumMethod(2, 3);
 System.out.println("sumResult = "+sumResult); // 8
 }
}
/*OUTPUT
sumResult = 8
*/

```

# OOP

## 1. SOLID principles

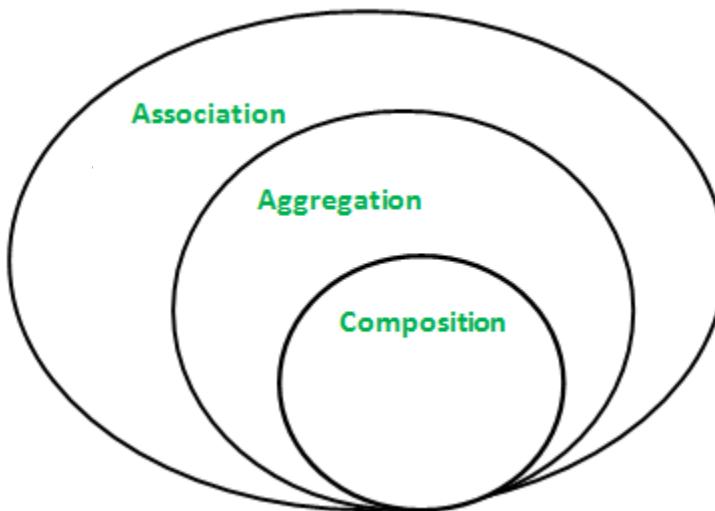


## 2. Association

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.



```
// Java program to illustrate the
// concept of Association

import java.io.*;

// class bank

class Bank {

 private String name;

 // bank name

 Bank(String name) {

 this.name = name;
 }

 public String getBankName() {

 return this.name;
 }
}

// employee class

class Employee{

 private String name;

 // employee name
```

```
Employee(String name) {
 this.name = name;
}

public String getEmployeeName() {
 return this.name;
}
}

// Association between both the
// classes in main method
class Association {

 public static void main (String[] args) {
 Bank bank = new Bank("Axis");
 Employee emp = new Employee("Neha");

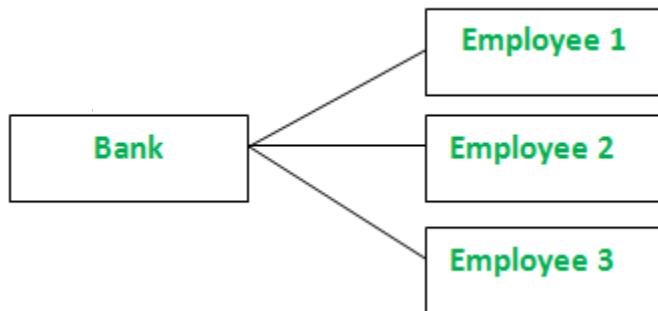
 System.out.println(emp.getEmployeeName() +
 " is employee of " + bank.getBankName());
 }
}
```

Run on IDE

Output:

```
Neha is employee of Axis
```

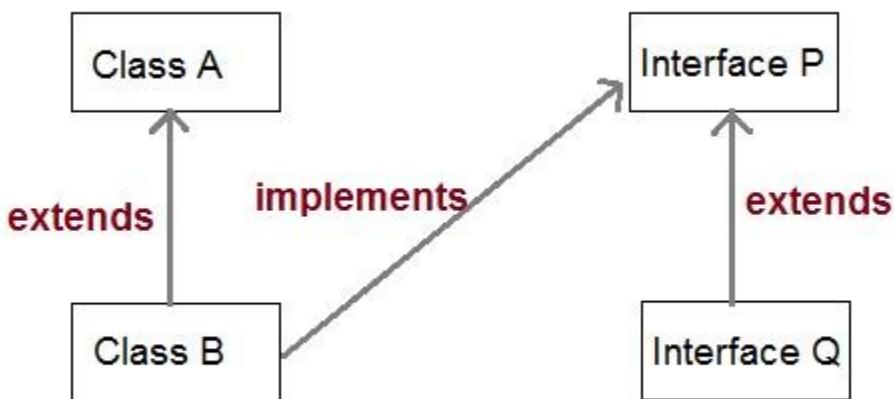
In above example two separate classes Bank and Employee are associated through their Objects. Bank can have many employees, So it is a one-to-many relationship.



## Inheritance

Inheritance is one of the key features of Object Oriented Programming. Inheritance provided mechanism that allowed **a class to inherit property of another class**. When a Class extends another class it inherits all non-private members including fields and methods. Inheritance in Java can be best understood in terms of Parent and Child relationship, also known as **Super class**(Parent) and **Sub class**(child) in Java language.

Inheritance defines **is-a** relationship between a Super class and its Sub class. **extends** and **implements** keywords are used to describe inheritance in Java.



Let us see how **extends** keyword is used to achieve Inheritance.

```

class Vehicle.
{

}
class Car extends Vehicle
{
```

```
..... //extends the property of vehicle class.
}
```

Now based on above example. In OOPs term we can say that,

- **Vehicle** is super class of **Car**.
  - **Car** is sub class of **Vehicle**.
  - Car IS-A Vehicle.
- 

### 2.1.1 Purpose of Inheritance

1. It promotes the code reusability i.e the same methods and variables which are defined in a parent/super/base class can be used in the child/sub/derived class.
  2. It promotes polymorphism by allowing method overriding.
- 

### 2.1.2 Disadvantages of Inheritance

Main disadvantage of using inheritance is that the two classes (parent and child class) gets **tightly coupled**.

This means that if we change code of parent class, it will affect to all the child classes which is inheriting/deriving the parent class, and hence, **it cannot be independent of each other**.

---

### 2.1.3 Simple example of Inheritance

```
class Parent
{
 public void p1()
 {
 System.out.println("Parent method");
 }
}
public class Child extends Parent {
 public void c1()
```

```

{
 System.out.println("Child method");
}
public static void main(String[] args)
{
 Child cobj = new Child();
 cobj.c1(); //method of Child class
 cobj.p1(); //method of Parent class
}
}

```

Child method

Parent method

## 2.1.4 Another example of Inheritance

```

class Vehicle
{
 String vehicleType;
}

public class Car extends Vehicle {

 String modelType;
 public void showDetail()
 {
 vehicleType = "Car"; //accessing Vehicle class member
 modelType = "sports";
 System.out.println(modelType+" "+vehicleType);
 }
 public static void main(String[] args)
 {
 Car car =new Car();
 car.showDetail();
 }
}

```

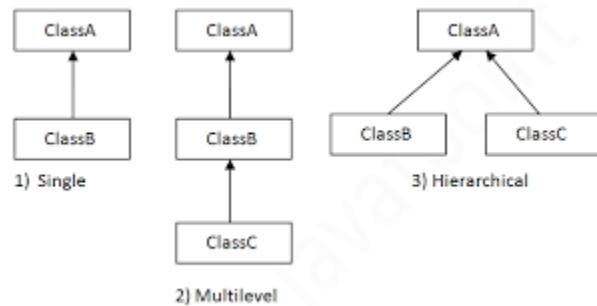
sports Car

---

## 2.1.5 Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Heirarchical Inheritance

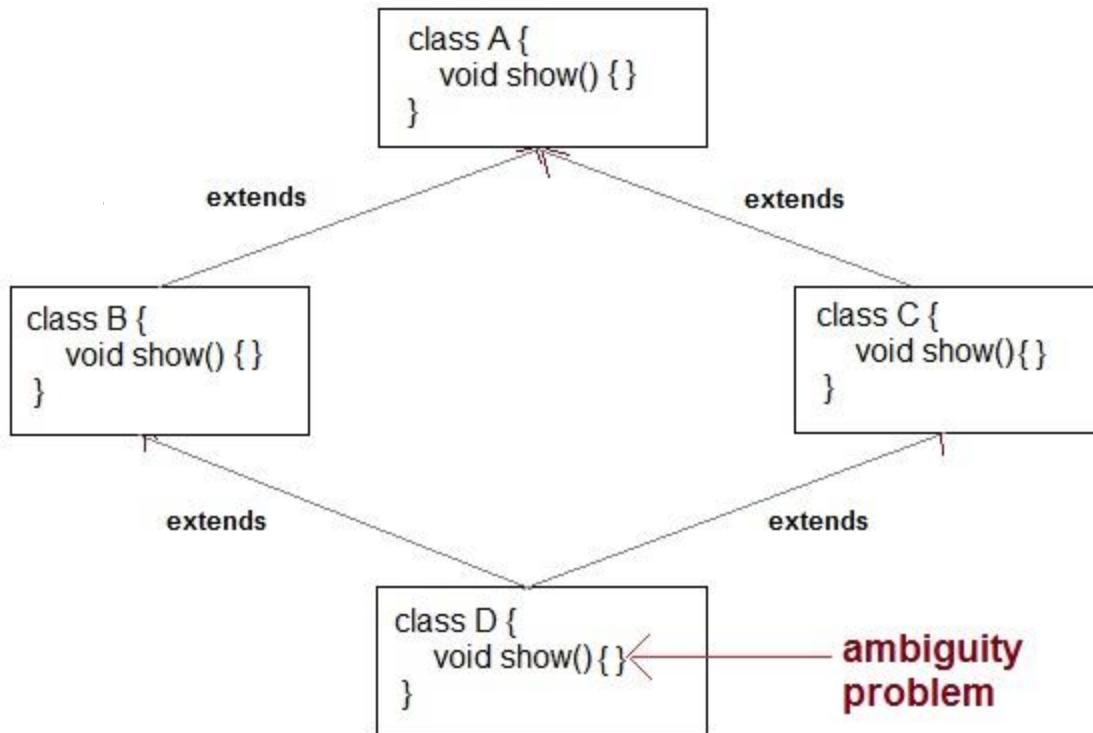
**NOTE :**Multiple inheritance is not supported in java



---

## 2.1.6 Why multiple inheritance is not supported in Java

- To remove ambiguity.
- To provide more maintainable and clear design.



## 2.1.7 super keyword

In Java, **super** keyword is used to refer to immediate parent class of a child class. In other words **super** keyword is used by a subclass whenever it need to refer to its immediate super class.

```

class Parent
{
 String name;
}
class Child extends Parent {

 String name;

 void detail()
 {
 super.name = "Parent";
 name = "Child";
 }
}

```

### 2.1.8 Example of Child class referring Parent class property using super keyword

```

class Parent
{
 String name;
}

public class Child extends Parent {
 String name;
 public void details()
 {
 super.name = "Parent"; //refers to parent class member
 name = "Child";
 System.out.println(super.name+" and "+name);
 }
 public static void main(String[] args)
 {
 Child cobj = new Child();
 cobj.details();
 }
}

```

```

 }
}
```

Parent and Child

---

## 2.1.9 Example of Child class referring Parent

### class methods using super keyword

```

class Parent
{
 String name;
 public void details()
 {
 name = "Parent";
 System.out.println(name);
 }
}

public class Child extends Parent {
 String name;
 public void details()
 {
 super.details(); //calling Parent class details() method
 name = "Child";
 System.out.println(name);
 }
 public static void main(String[] args)
 {
 Child cobj = new Child();
 cobj.details();
 }
}
```

Parent

Child

---

## 2.1.10 Example of Child class calling Parent class constructor using super keyword

```

class Parent
{
 String name;

 public Parent(String n)
 {
 name = n;
 }

}

public class Child extends Parent {
 String name;

 public Child(String n1, String n2)
 {

 super(n1); //passing argument to parent class constructor
 this.name = n2;
 }

 public void details()
 {
 System.out.println(super.name+" and "+name);
 }

 public static void main(String[] args)
 {
 Child cobj = new Child("Parent","Child");
 cobj.details();
 }
}

```

Parent and Child

**Note:** When calling the parent class constructor from the child class using super keyword, super keyword should always be the first line in the method/constructor of the child class.

---

### 2.1.11 Super class reference pointing to Sub class object.

In context to above example where Class B extends class A.

```
A a=new B();
```

is legal syntax because of IS-A relationship is there between class A and Class B.

---

### 2.1.12 Q. Can you use both this() and super() in a Constructor?

NO, because both super() and this() must be first statement inside a constructor. Hence we cannot use them together.

## Aggregation

It 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 effect the other entity

```
// Java program to illustrate
```

```
//the concept of Aggregation.
```

```
import java.io.*;
import java.util.*;
```

```
// student class
```

```
class Student {
 String name;
 int id ;
 String dept;
```

```
Student(String name, int id, String dept) {
 this.name = name;
 this.id = id;
 this.dept = dept;
}

/* Department class contains list of student
Objects. It is associated with student
class through its Object(s). */

class Department {
 String name;
 private List<Student> students;
 Department(String name, List<Student>
 students) {
 this.name = name;
 this.students = students;
 }

 public List<Student> getStudents() {
 return students;
 }
}

/* Institute class contains list of Department
Objects. It is asoociated with Department
class through its Object(s).*/
```

```
class Institute {
 String instituteName;
 private List<Department> departments;

 Institute(String instituteName, List<Department>
 departments) {
 this.instituteName = instituteName;
 this.departments = departments;
 }

 // count total students of all departments
 // in a given institute
 public int getTotalStudentsInInstitute()
 {
 int noOfStudents = 0;
 List<Student> students;
 for(Department dept : departments) {
 students = dept.getStudents();
 for(Student s : students) {
 noOfStudents++;
 }
 }
 return noOfStudents;
 }
}

// main method
class GFG{
 public static void main (String[] args) {
```

```
Student s1 = new Student("Mia", 1, "CSE");
Student s2 = new Student("Priya", 2, "CSE");
Student s3 = new Student("John", 1, "EE");
Student s4 = new Student("Rahul", 2, "EE");

// making a List of
// CSE Students.

List <Student> cse_students = new ArrayList<Student>();
cse_students.add(s1);
cse_students.add(s2);

// making a List of
// EE Students

List <Student> ee_students = new ArrayList<Student>();
ee_students.add(s3);
ee_students.add(s4);

Department CSE = new Department("CSE", cse_students);
Department EE = new Department("EE", ee_students);

List <Department> departments = new
ArrayList<Department>();
departments.add(CSE);
departments.add(EE);

// creating an instance of Institute.

Institute institute = new Institute("BITS", departments);
```

```

 System.out.print("Total students in institute: ");
 System.out.print(institute.getTotalStudentsInInstitute()
);
}

}

```

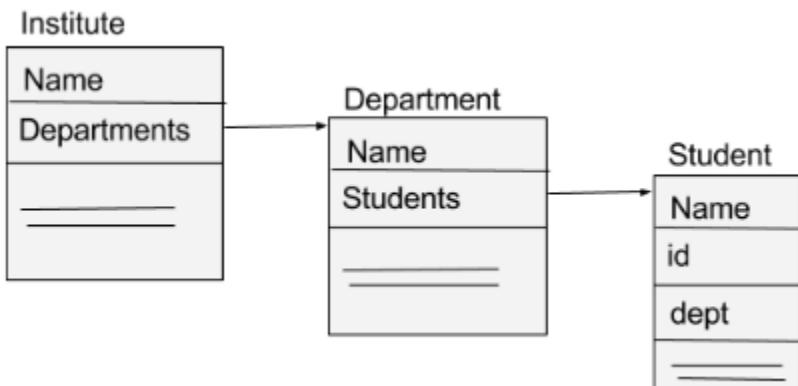
Run on IDE

Output:

```
Total students in institute: 4
```

In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students. So, we make a Institute class which has a reference to Object or no. of Objects (i.e. List of Objects) of the Department class. That means Institute class is associated with Department class through its Object(s). And Department class has also a reference to Object or Objects (i.e. List of Objects) of Student class means it is associated with Student class through its Object(s).

It represents a **Has-A** relationship.



### When do we use Aggregation ??

Code reuse is best achieved by aggregation.

## Composition

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.

Lets take example of **Library**.

```

// Java program to illustrate

// the concept of Composition

import java.io.*;
import java.util.*;

```

```
// class book

class Book {
 public String title;
 public String author;
 Book(String title, String author) {
 this.title = title;
 this.author = author;
 }
}

// Library class contains
// list of books.

class Library {
 // reference to refer to list of books.
 private final List<Book> books;

 Library (List<Book> books) {
 this.books = books;
 }

 public List<Book> getTotalBooksInLibrary() {
 return books;
 }
}

// main method

class GFG
{
 public static void main (String[] args) {
```

```

 // Creating the Objects of Book class.

 Book b1 = new Book("EffectiveJ Java", "Joshua
Bloch");

 Book b2 = new Book("Thinking in Java", "Bruce
Eckel");

 Book b3 = new Book("Java: The Complete Reference",
"Herbert Schildt");

 // Creating the list which contains the
 // no. of books.

 List<Book> books = new ArrayList<Book>();
 books.add(b1);
 books.add(b2);
 books.add(b3);

 Library library = new Library(books);

 List<Book> bks = library.getTotalBooksInLibrary();
 for(Book bk : bks) {

 System.out.println("Title : " + bk.title + " and
"
 +" Author : " + bk.author);
 }
}

```

**Run on IDE**

**Output**

```
Title : EffectiveJ Java and Author : Joshua Bloch
Title : Thinking in Java and Author : Bruce Eckel
```

Title : Java: The Complete Reference and Author : Herbert Schildt

In above example a library can have no. of **books** on same or different subjects. So, If Library gets destroyed then All books within that particular library will be destroyed. i.e. book can not exist without library. That's why it is composition.

### Aggregation vs Composition

1. **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
2. **Type of Relationship:** Aggregation relation is “**has-a**” and composition is “**part-of**” relation.
3. **Type of association:** Composition is a **strong** Association whereas Aggregation is a **weak** Association.

```
// Java program to illustrate the
// difference between Aggregation
// Composition.
```

```
import java.io.*;

// Engine class which will
// be used by car. so 'Car'
// class will have a field
// of Engine type.

class Engine
{
 // starting an engine.

 public void work()
 {

 System.out.println("Engine of car has been started");
 }
}
```

```
}

}

// Engine class
final class Car
{

 // For a car to move,
 // it need to have a engine.
 private final Engine engine; // Composition
 //private Engine engine; // Aggregation

 Car(Engine engine)
 {
 this.engine = engine;
 }

 // car start moving by starting engine
 public void move()
 {

 //if(engine != null)
 {
 engine.work();
 System.out.println("Car is moving ");
 }
 }
}
```

```

}

class GFG
{
 public static void main (String[] args)
 {

 // making an engine by creating
 // an instance of Engine class.
 Engine engine = new Engine();

 // Making a car with engine.
 // so we are passing a engine
 // instance as an argument while
 // creating instace of Car.
 Car car = new Car(engine);
 car.move();

 }
}

```

**Run on IDE**

**Output:**

```

Engine of car has been started
Car is moving

```

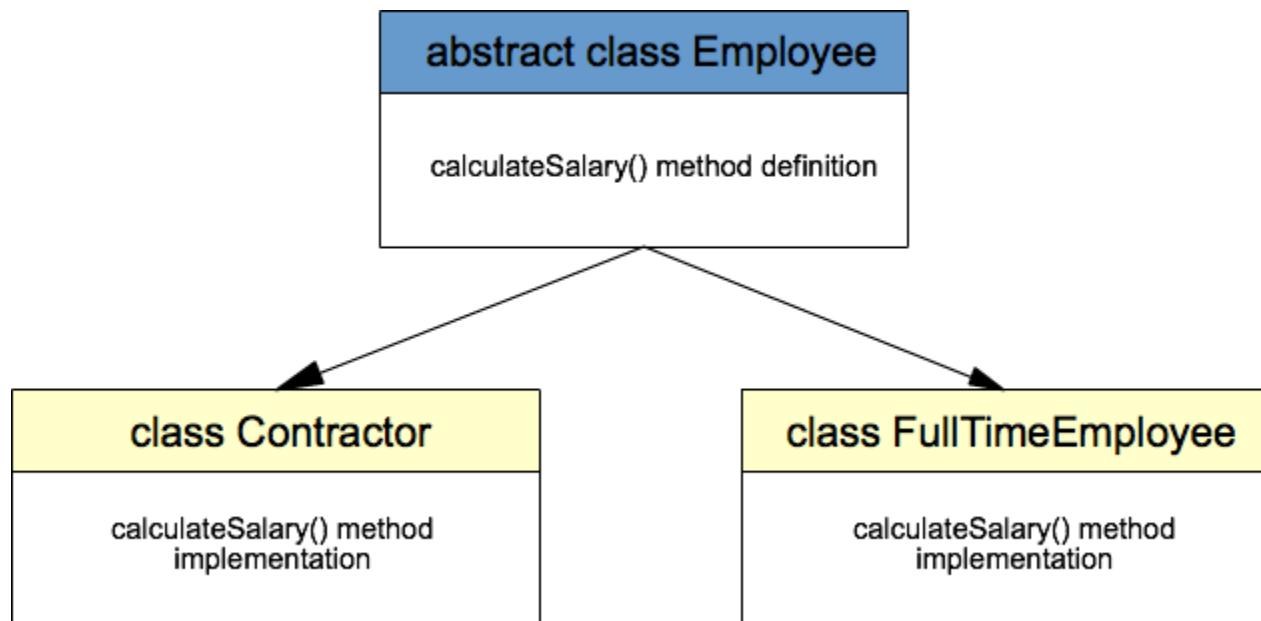
In case of aggregation, the Car also performs its functions through an Engine, but the Engine is not always an internal part of the Car. An engine can be swapped out or even can be removed from the car. That's why we make The Engine type field non-final.

# What is Abstraction

Abstraction is a process of hiding the implementation details from the user. Only the functionality will be provided to the user. In Java, abstraction is achieved using abstract classes and interfaces.

## Java Abstraction Example

To give an example of abstraction we will create one superclass called Employee and two subclasses – Contractor and FullTimeEmployee. Both subclasses have common properties to share, like the name of the employee and the amount of money the person will be paid per hour. There is one major difference between contractors and full-time employees – the time they work for the company. Full-time employees work constantly 8 hours per day and the working time of contractors may vary.



Java abstract class example

Lets first create the superclass Employee. Note the usage of **abstract** keyword in class definition. This marks the class to be abstract, which means it can not be instantiated directly. We define a method called `calculateSalary()` as an abstract method. This way you leave the implementation of this method to the inheritors of the Employee class.

```
package net.javatutorial;
```

```
public abstract class Employee {
```

```

private String name;
private int paymentPerHour;

public Employee(String name, int paymentPerHour) {
 this.name = name;
 this.paymentPerHour = paymentPerHour;
}

public abstract int calculateSalary();
public String getName() {
 return name;
}

public void setName(String name) {
 this.name = name;
}

public int getPaymentPerHour() {
 return paymentPerHour;
}

public void setPaymentPerHour(int paymentPerHour) {
 this.paymentPerHour = paymentPerHour;
}
}

```

The Contractor class inherits all properties from its parent Employee but have to provide it's own implementation to calculateSalary() method. In this case we multiply the value of payment per hour with given working hours.

```
package net.javatutorial;
```

```
public class Contractor extends Employee {
 private int workingHours;
 public Contractor(String name, int paymentPerHour, int workingHours) {
 super(name, paymentPerHour);
 this.workingHours = workingHours;
 }
}
```

**@Override**

```
public int calculateSalary() {
 return getPaymentPerHour() * workingHours;
}
}
```

The FullTimeEmployee also has its own implementation of calculateSalary() method. In this case we just multiply by constant 8 hours.

```
package net.javatutorial;
public class FullTimeEmployee extends Employee {
 public FullTimeEmployee(String name, int paymentPerHour) {
 super(name, paymentPerHour);
 }
}
```

**@Override**

```
public int calculateSalary() {
 return getPaymentPerHour() * 8;
}
}
```

## What is 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.

### Example

Following is an example that demonstrates how to achieve Encapsulation in Java –

```
/* File name : EncapTest.java */

public class EncapTest {

 private String name;
 private String idNum;
 private int age;

 public int getAge() {
 return age;
 }

 public String getName() {
 return name;
 }

 public String getIdNum() {
 return idNum;
 }

 public void setAge(int newAge) {
 age = newAge;
 }
}
```

```

}

public void setName(String newName) {
 name = newName;
}

public void setIdNum(String newId) {
 idNum = newId;
}

}

```

The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class. Normally, these methods are referred as getters and setters. Therefore, any class that wants to access the variables should access them through these getters and setters.

The variables of the EncapTest class can be accessed using the following program –

```

/* File name : RunEncap.java */

public class RunEncap {

 public static void main(String args[]) {
 EncapTest encap = new EncapTest();
 encap.setName("James");
 encap.setAge(20);
 encap.setIdNum("12343ms");

 System.out.print("Name : " + encap.getName() + " Age : " +
encap.getAge());
 }
}

```

This will produce the following result –

### **Output**

Name : James Age : 20

### **Benefits of Encapsulation**

The fields of a class can be made read-only or write-only.

A class can have total control over what is stored in its fields.

## **Difference between Abstraction and Encapsulation ?**

|                         | <b>Abstraction</b>                                                                                                                    | <b>Encapsulation</b>                                                                                                            |
|-------------------------|---------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------|
| Short description       | Abstraction is a process, which extracts the essential details about an item, or group of items, and ignores the inessential details. | Encapsulation is a process which wraps or encloses the data in a capsule or makes the data concise.                             |
| They are                | Abstraction relates to the idea of hiding data that is not needed for presentation.                                                   | Encapsulation is grouping together of data and functionality.                                                                   |
| Basic functions         | Basically, abstraction is used for hiding the unwanted data and it gives the relevant data.                                           | Basically, encapsulation means hiding the code and data into a single unit in order to protect the data from the outside world. |
| When are they Operated  | It operates the problem in the design level.                                                                                          | It operates the problem in the implementation level.                                                                            |
| Beneficial to the roles | It helps the user to focus on what the object does instead of how it does any function.                                               | It helps the programmer in hiding the internal details or mechanics of how an object does something.                            |

|                |                                                                                                   |                                                                                                                                       |
|----------------|---------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| Information is | Here information is separated from the real data.                                                 | Here information is wrapped in a hidden format.                                                                                       |
| Layouts        | It is concerned with the outer layout, which is used in terms of design.                          | It is concerned with the inner layout, which is used in terms of implementation.                                                      |
| For Example    | The Outer Look of a Mobile Phone, which has a display screen and keypad buttons to dial a number. | Inner Implementation detail of a Mobile Phone, how the keypad button and Display Screen are connected with each other using circuits. |

## Difference between Data hiding and Abstraction?

### Data Hiding

1. The data should not go out directly i.e outside person is not allowed to access the data this is nothing but “**Data Hiding**”.
2. The main advantage of data hiding is we can achieve security.
3. By using **private** modifier we can achieve this.

E.g

```

1. class Employee {
2. private int empId;
3. private String empName;
4. private Date empDOB;
5. ...
6. ...
7. }
```

### Note

It is highly recommended to declare data members with private modifier.

### Data Abstraction

1. Hiding implementation details is nothing but abstraction. The main advantages of abstraction are we can achieve security as we are not highlighting internal implementation
2. Enhancement will become easy. With out effecting outside person we can change our internal implementation.
3. It improves maintainability.
4. By using interfaces and abstract classes we can achieve abstraction.

## Note

1. If we don't know about implementation just we have to represent the specification then we should go for **interface**.
2. If we don't know about complete implementation just we have partial implementation then we should go for **abstract**.
3. If we know complete implementation and if we r ready to provide service then we should go for **concrete class**

## Polymorphism

### 2. Rules for method overriding:

#### 2.2.1 Overriding and Access-Modifiers :

The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

```
// A Simple Java program to demonstrate
// Overriding and Access-Modifiers

class Parent
{
 // private methods are not overridden
 private void m1() { System.out.println("From parent
m1()"); }

 protected void m2() { System.out.println("From parent
m2()"); }
}

class Child extends Parent
{
 // new m1() method
 // unique to Child class
```

```

 private void m1() { System.out.println("From child
m1()"); }

 // overriding method
 // with more accessibility
 @Override

 public void m2() { System.out.println("From child
m2()"); }

}

// Driver class
class Main
{
 public static void main(String[] args)
 {
 Parent obj1 = new Parent();
 obj1.m2();

 Parent obj2 = new Child();
 obj2.m2();
 }
}

```

**Output :**

```

From parent m2()
From child m2()

```

## 2.2.2Final methods can not be overridden

If we don't want a method to be overridden, we declare it as final. Please see [Using final with Inheritance](#).

```
// A Java program to demonstrate that
// final methods cannot be overridden
```

```
class Parent
{
 // Can't be overridden
 final void show() { }
}
```

```
class Child extends Parent
{
 // This would produce error
 void show() { }
}
```

Output:

```
13: error: show() in Child cannot override show() in Parent
 void show() { }
 ^
overridden method is final
```

### **2.2.3 Static methods can not be overridden(Method Overriding vs Method Hiding)**

When you defines a static method with same signature as a static method in base class, it is known as method hiding.

The following table summarizes what happens when you define a method with the same signature as a method in a super-class.

|                 | <b>SUPERCLASS</b> | <b>SUPERCLASS</b>    |
|-----------------|-------------------|----------------------|
|                 | <b>INSTANCE</b>   | <b>STATIC METHOD</b> |
|                 | <b>METHOD</b>     |                      |
| <b>SUBCLASS</b> | Overrides         | Generates a          |
| <b>INSTANCE</b> |                   | compile-time         |
| <b>METHOD</b>   |                   | error                |
| <b>SUBCLASS</b> | Generates a       | Hides                |
| <b>STATIC</b>   | compile-time      |                      |
| <b>METHOD</b>   | error             |                      |

```
/*
Java program to show that if static method is redefined by
a derived class, then it is not overriding,it is hiding */

class Parent
{
 // Static method in base class which will be hidden in subclass
 static void m1() { System.out.println("From parent static m1()"); }

 // Non-static method which will be overridden in derived class
 void m2() { System.out.println("From parent non-static(instance) m2()"); }
}

class Child extends Parent
{
 // This method hides m1() in Parent
 static void m1() { System.out.println("From child static m1()"); }
}
```

```
// This method overrides m2() in Parent
@Override
 public void m2() { System.out.println("From child non-static(instance) m2()"); }

}

// Driver class
class Main
{
 public static void main(String[] args)
 {
 Parent obj1 = new Child();

 // As per overriding rules this should call to class Child
 static
 // overridden method. Since static method can not be overridden,
 it
 // calls Parent's m1()
 obj1.m1();

 // Here overriding works and Child's m2() is called
 obj1.m2();
 }
}
```

Output:

```
From parent static m1()
From child non-static(instance) m2()
```

#### 2.2.4 **Private methods can not be overridden**

Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.

#### 2.2.5 **The overriding method must have same return type (or subtype)**

From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as **covariant return type**.

#### 2.2.6 **Overriding and constructor:**

We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).

### **2.2.7 Overriding and Exception-Handling**

Below are two rules to note when overriding methods related to exception-handling.

- **Rule#1 :** If the super-class overridden method does not throws an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

```
/* Java program to demonstrate overriding when
superclass method does not declare an exception

*/
class Parent
{
 void m1() { System.out.println("From parent m1()"); }

 void m2() { System.out.println("From parent m2()"); }
}

class Child extends Parent
```

```

{

 @Override
 // no issue while throwing unchecked exception
 void m1() throws ArithmeticException
 { System.out.println("From child m1()"); }

 @Override
 // compile-time error
 // issue while throwin checked exception
 void m2() throws Exception{ System.out.println("From child
m2"); }

}

```

- Output:

- error: m2() in Child cannot override m2() in Parent
- void m2() throws Exception{ System.out.println("From child m2");}
- ^
- overridden method does not throw Exception

- **Rule#2 :**If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in [Exception hierarchy](#) will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

/\* Java program to demonstrate overriding  
when

superclass method does declare an  
exception

\*/

```

class Parent
{

```

```
void m1() throws RuntimeException
{
 System.out.println("From parent
m1()");}
```

}

```
class Child1 extends Parent
{
 @Override
 // no issue while throwing same
exception
 void m1() throws RuntimeException
 {
 System.out.println("From child1
m1()");}
```

}

```
class Child2 extends Parent
{
 @Override
 // no issue while throwing subclass
exception
 void m1() throws ArithmeticException
 {
 System.out.println("From child2
m1()");}
```

}

```
class Child3 extends Parent
{
 @Override
```

```

 // no issue while not throwing any
 exception

 void m1()
 {
 System.out.println("From child3
m1()"); }

 }

class Child4 extends Parent
{
 @Override
 // compile-time error
 // issue while throwing parent
 exception

 void m1() throws Exception
 {
 System.out.println("From child4
m1()"); }

}

```

- Output:

- error: m1() in Child4 cannot override m1() in Parent
- void m1() throws Exception
- ^
- overridden method does not throw Exception

### 2.2.8 Overriding and abstract method

Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise compile-time error will be thrown.

### 2.2.9 Overriding and synchronized/stricfp method

- The presence of synchronized/stricfp modifier with method have no effect on the rules of overriding, i.e. it's possible that a synchronized/stricfp method can override a non synchronized/stricfp one and vice-versa.

# Design Patterns

**Design Patterns** are very popular among software developers. A design pattern is a well-described solution to a common software problem.

Some of the benefits of using design patterns are:

- Design Patterns are already defined and provides industry standard approach to solve a recurring problem, so it saves time if we sensibly use the design pattern.
- Using design patterns promotes reusability that leads to more robust and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.
- Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

**Java Design Patterns** are divided into three categories – **creational**, **structural**, and **behavioral** design patterns.

## 1. Creational Design Patterns

Creational design patterns provide solution to instantiate an object in the best possible way for specific situations.

The basic form of object creation could result in design problems or add unwanted complexity to the design. Creational design patterns solve this problem by controlling the object creation by different ways.

There are five creational design patterns that we will discuss in this eBook.

1. Singleton Pattern
2. Factory Pattern
3. Abstract Factory Pattern
4. Builder Pattern
5. Prototype Pattern

All these patterns solve specific problems with object creation, so you should understand and use them when needed.

## 1. Singleton Pattern

**Singleton** is one of the **Gangs of Four Design patterns** and comes in the **Creational Design Pattern** category. From the definition, it seems to be a very simple design pattern but when it comes to implementation, it comes with a lot of implementation concerns. The implementation of Singleton pattern has always been a controversial topic among developers. Here we will learn about Singleton design pattern principles, different ways to implement Singleton and some of the best practices for its usage. 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`. To implement Singleton pattern, we have different approaches but all of them have following common concepts.

- Private constructor to restrict instantiation of the class from other classes.
- Private static variable of the same class that is the only instance of the class.
- Public static method that returns the instance of the class, this is the global access point for outer world to get the instance of the singleton class.

In further sections, we will learn different approaches of Singleton pattern implementation and design concerns with the implementation.

### 1.1.1 Eager Initialization

In eager initialization, the instance of Singleton Class is created at the time of class loading, this is the easiest method to create a singleton class but it has a drawback that instance is created even though client application might not be using it.

Here is the implementation of static initialization singleton class.

```
package com.journaldev.singleton;
public class EagerInitializedSingleton {
 private static final EagerInitializedSingleton instance = new
```

```

EagerInitializedSingleton();
//private constructor to avoid client applications to use
constructor
private EagerInitializedSingleton(){}
public static EagerInitializedSingleton getInstance(){
 return instance;
}
}

```

If your singleton class is not using a lot of resources, this is the approach to use. But in most of the scenarios, Singleton classes are created for resources such as File System, Database connections etc and we should avoid the instantiation until unless client calls the *getInstance* method. Also this method doesn't provide any options for exception handling.

### 1.1.2 Static block initialization

Static block initialization implementation is similar to eager initialization, except that instance of class is created in the static block that provides option for exception handling.

```

package com.journaldev.singleton;
public class StaticBlockSingleton {
 private static StaticBlockSingleton instance;
 private StaticBlockSingleton(){}
 //static block initialization for exception handling
 static{
 try{
 instance = new StaticBlockSingleton();
 }catch(Exception e){
 throw new RuntimeException("Exception occurred in creating
singleton instance");
 }
 }
 public static StaticBlockSingleton getInstance(){
 return instance;
 }
}

```

Both eager initialization and static block initialization creates the instance even before it's being used and that is not the best practice to use. So in further sections, we will learn how to create Singleton class that supports lazy initialization.

### 1.1.3 Lazy Initialization

Lazy initialization method to implement Singleton pattern creates the instance in the global access method. Here is the sample code for creating Singleton class with this

```

package com.journaldev.singleton;
public class LazyInitializedSingleton {

```

```

private static LazyInitializedSingleton instance;
private LazyInitializedSingleton(){}
public static LazyInitializedSingleton getInstance(){
if(instance == null){
instance = new LazyInitializedSingleton();
}
return instance;
}
}
}

```

The above implementation works fine in case of single threaded environment but when it comes to multithreaded systems, it can cause issues

if multiple threads are inside the if loop at the same time. It will destroy the singleton pattern and both threads will get the different instances of singleton class. In next section, we will see different ways to create a **threadsafe** singleton class.

#### 1.1.4 Thread Safe Singleton

The easier way to create a thread-safe singleton class is to make the global access method **synchronized**, so that only one thread can execute this method at a time. General implementation of this approach is like the below class.

```

package com.journaldev.singleton;
public class ThreadSafeSingleton {
private static ThreadSafeSingleton instance;
private ThreadSafeSingleton(){}
public static synchronized ThreadSafeSingleton getInstance(){
if(instance == null){
instance = new ThreadSafeSingleton();
}
return instance;
}
}

```

Above implementation works fine and provides thread-safety but it reduces the performance because of cost associated with the synchronized method, although we need it only for the first few threads who might create the separate instances (Read: [Java Synchronization](#)). To avoid this extra overhead every time, **double checked locking** principle is used. In this approach, the synchronized block is used inside if condition with an additional check to ensure that only one instance of singleton class is created.

Below code snippet provides the double checked locking implementation.

```

public static ThreadSafeSingleton getInstanceUsingDoubleLocking(){
if(instance == null){
synchronized (ThreadSafeSingleton.class) {
if(instance == null){

```

```

instance = new ThreadSafeSingleton();
}
}
}
return instance;
}

```

### 1.1.5 Bill Pugh Singleton Implementation

Prior to Java 5, java memory model had a lot of issues and above approaches

used to fail in certain scenarios where too many threads try to get the instance of the Singleton class simultaneously. So Bill Pugh came up with a different approach to create the Singleton class using an **inner static helper class**. The Bill Pugh Singleton implementation goes like this;

```

package com.journaldev.singleton;
public class BillPughSingleton {
private BillPughSingleton(){}
private static class SingletonHelper{
private static final BillPughSingleton INSTANCE = new
BillPughSingleton();
}
public static BillPughSingleton getInstance(){
return SingletonHelper.INSTANCE;
}
}

```

Notice the **private inner static class** that contains the instance of the singleton class. When the singleton class is loaded, SingletonHelper class is

not loaded into memory and only when someone calls the *getInstance* method, this class gets loaded and creates the Singleton class instance. This is the most widely used approach for Singleton class as it doesn't require synchronization. I am using this approach in many of my projects and it's easy to understand and implement also.

### 1.1.6 Using Reflection to destroy Singleton Pattern

Reflection can be used to destroy all the above singleton implementation approaches. Let's see this with an example class.

```

package com.journaldev.singleton;
import java.lang.reflect.Constructor;
public class ReflectionSingletonTest {
public static void main(String[] args) {
EagerInitializedSingleton instanceOne =
EagerInitializedSingleton.getInstance();
EagerInitializedSingleton instanceTwo = null;
try {
Constructor[] constructors =
EagerInitializedSingleton.class.getDeclaredConstructors();
for (Constructor constructor : constructors) {
//Below code will destroy the singleton pattern
}
}
}

```

```

constructor.setAccessible(true);
instanceTwo = (EagerInitializedSingleton)
constructor.newInstance();
break;
}
} catch (Exception e) {
e.printStackTrace();
}
System.out.println(instanceOne.hashCode());
System.out.println(instanceTwo.hashCode());
}
}

```

When you run the above test class, you will notice that hashCode of both the instances are not same that destroys the singleton pattern. Reflection is very powerful and used in a lot of frameworks like Spring and Hibernate, do check out [Java Reflection Tutorial](#).

### 1.1.7Enum Singleton

To overcome this situation with Reflection, Joshua Bloch suggests the use of

Enum to implement Singleton design pattern as Java ensures that any enum

value is instantiated only once in a Java program. Since [Java Enum](#) values are globally accessible, so is the singleton. The drawback is that the enum type is somewhat inflexible; for example, it does not allow lazy initialization.

```

package com.journaldev.singleton;
public enum EnumSingleton {
INSTANCE;
public static void doSomething(){
//do something
}
}

```

### 1.1.8Serialization and Singleton

Sometimes in distributed systems, we need to implement Serializable interface in Singleton class so that we can store its state in file system and retrieve it at later point of time. Here is a small singleton class that implements Serializable interface also.

```

package com.journaldev.singleton;
import java.io.Serializable;
public class SerializedSingleton implements Serializable{
private static final long serialVersionUID = -7604766932017737115L;
private SerializedSingleton(){}
private static class SingletonHelper{
private static final SerializedSingleton instance = new

```

```

SerializedSingleton();
}
public static SerializedSingleton getInstance(){
return SingletonHelper.instance;
}
}

```

The problem with above serialized singleton class is that whenever we deserialize it, it will create a new instance of the class. Let's see it with a simple program.

```

package com.journaldev.singleton;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
public class SingletonSerializedTest {
public static void main(String[] args) throws
FileNotFoundException, IOException, ClassNotFoundException {
SerializedSingleton instanceOne =
SerializedSingleton.getInstance();
ObjectOutput out = new ObjectOutputStream(new FileOutputStream(
"filename.ser"));
out.writeObject(instanceOne);
out.close();
//deserialze from file to object
ObjectInput in = new ObjectInputStream(new FileInputStream(
"filename.ser"));
SerializedSingleton instanceTwo = (SerializedSingleton)
in.readObject();
in.close();
System.out.println("instanceOne
hashCode="+instanceOne.hashCode());
System.out.println("instanceTwo
hashCode="+instanceTwo.hashCode());
}
}

```

Output of the above program is;

```

instanceOne hashCode=2011117821
instanceTwo hashCode=109647522

```

So it destroys the singleton pattern, to overcome this scenario all we need to

do it provide the implementation of readResolve() method.

```

protected Object readResolve() {
return getInstance();
}

```

After this you will notice that hashCode of both the instances are same in test program.

## 2. Factory Pattern

**Factory Pattern** is one of the **Creational Design pattern** and it's widely used in JDK as well as frameworks like Spring and Struts. Factory design pattern is used when we have a super class with multiple subclasses and based on input, we need to return one of the sub-class. This pattern take out the responsibility of instantiation of a class from client program to the factory class. Let's first learn how to implement factory pattern in java and then we will learn its benefits and we will see its usage in JDK.

### 1.2.1 Super Class

Super class in factory pattern can be an interface, **abstract class** or a normal java class. For our example, we have super class as abstract class with **overridden** `toString()` method for testing purpose.

```
package com.journaldev.design.model;
public abstract class Computer {
 public abstract String getRAM();
 public abstract String getHDD();
 public abstract String getCPU();
 @Override
 public String toString(){
 return "RAM= "+this.getRAM()+", HDD="+this.getHDD()+", CPU="+this.getCPU();
 }
}
```

### 1.2.2 Sub Classes

Let's say we have two sub-classes PC and Server with below implementation.

```
package com.journaldev.design.model;
public class PC extends Computer {
 private String ram;
 private String hdd;
 private String cpu;
 public PC(String ram, String hdd, String cpu){
 this.ram=ram;
 this.hdd=hdd;
 this.cpu=cpu;
 }
 @Override
 public String getRAM() {
 return this.ram;
 }
 @Override
 public String getHDD() {
 return this.hdd;
 }
 @Override
 public String getCPU() {
```

```

 return this.cpu;
 }
}

```

Notice that both the classes are extending Computer class.

```

package com.journaldev.design.model;
public class Server extends Computer {
 private String ram;
 private String hdd;
 private String cpu;
 public Server(String ram, String hdd, String cpu){
 this.ram=ram;
 this.hdd=hdd;
 this.cpu=cpu;
 }
 @Override
 public String getRAM() {
 return this.ram;
 }
 @Override
 public String getHDD() {
 return this.hdd;
 }
 @Override
 public String getCPU() {
 return this.cpu;
 }
}

```

### 1.2.3 Factory Class

Now that we have super classes and sub-classes ready, we can write our factory class. Here is the basic implementation.

```

package com.journaldev.design.factory;
import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;
import com.journaldev.design.model.Server;
public class ComputerFactory {
 public static Computer getComputer(String type, String ram, String hdd, String cpu){
 if("PC".equalsIgnoreCase(type)) return new PC(ram, hdd, cpu);
 else if("Server".equalsIgnoreCase(type)) return new Server(ram,hdd, cpu);
 return null;
 }
}

```

1. We can keep Factory class **Singleton** or we can keep the method that returns the subclass as **static**.
2. Notice that based on the input parameter, different subclass is created and returned.

Here is a simple test client program that uses above factory pattern implementation.

```

package com.journaldev.design.test;
import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;
public class TestFactory {
 public static void main(String[] args) {
 Computer pc = ComputerFactory.getComputer("pc","2 GB","500GB","2.4 GHz");
 Computer server = ComputerFactory.getComputer("server","16GB","1 TB","2.9 GHz");
 System.out.println("Factory PC Config::"+pc);
 System.out.println("Factory Server Config::"+server);
 }
}

```

Output of above program is:

Factory PC **Config**::RAM= **2** GB, HDD=**500** GB, CPU=**2.4** GHz  
 Factory Server **Config**::RAM= **16** GB, HDD=**1** TB, CPU=**2.9** GHz

#### 1.2.4 Benefits of Factory Pattern

1. Factory pattern provides approach to code for interface rather than implementation.
2. Factory pattern removes the instantiation of actual implementation classes from client code, making it more robust, less coupled and easy to extend. For example, we can easily change PC class implementation because client program is unaware of this.
3. Factory pattern provides abstraction between implementation and client classes through inheritance.

#### 1.2.5 Factory Pattern Examples in JDK

1. `java.util.Calendar`, `ResourceBundle` and `NumberFormat getInstance()` methods uses Factory pattern.
2. `valueOf()` method in wrapper classes like `Boolean`, `Integer` etc.

### 3. Abstract Factory Pattern

Abstract Factory is one of the **Creational pattern** and almost similar to **Factory Pattern** except the fact that it's more like factory of factories.

If you are familiar with **factory design pattern in java**, you will notice that we have a single Factory class that returns the different sub-classes based on the input provided and factory class uses if-else or switch statement to achieve this.

In Abstract Factory pattern, we get rid of if-else block and have a factory class for each sub-class and then an Abstract Factory class that will return the sub-class based on the input factory class. At first it seems confusing but once you see the implementation, it's really easy to grasp and

understand the minor difference between Factory and Abstract Factory pattern.

Like our factory pattern post, we will use the same super class and subclasses.

### 1.3.1 Super Class and Sub-Classes

```
package com.journaldev.design.model;
public abstract class Computer {
 public abstract String getRAM();
 public abstract String getHDD();
 public abstract String getCPU();
 @Override
 public String toString(){
 return "RAM= "+this.getRAM()+" , HDD= "+this.getHDD()+" , CPU= "+this.getCPU();
 }
}
package com.journaldev.design.model;
public class PC extends Computer {
 private String ram;
 private String hdd;
 private String cpu;
 public PC(String ram, String hdd, String cpu){
 this.ram=ram;
 this.hdd=hdd;
 this.cpu=cpu;
 }
 @Override
 public String getRAM() {
 return this.ram;
 }
 @Override
 public String getHDD() {
 return this.hdd;
 }
 @Override
 public String getCPU() {
 return this.cpu;
 }
}
package com.journaldev.design.model;
public class Server extends Computer {
 private String ram;
 private String hdd;
 private String cpu;
 public Server(String ram, String hdd, String cpu){
 this.ram=ram;
 this.hdd=hdd;
 this.cpu=cpu;
 }
 @Override
 public String getRAM() {
 return this.ram;
 }
 @Override
```

```

public String getHDD() {
 return this.hdd;
}
@Override
public String getCPU() {
 return this.cpu;
}
}

```

### 1.3.2 Factory Classes for Each sub-class

First of all we need to create an Abstract Factory interface or **abstract class**.

```

package com.journaldev.design.abstractfactory;
import com.journaldev.design.model.Computer;
public interface ComputerAbstractFactory {
 public Computer createComputer();
}

```

Notice that *createComputer()* method is returning an instance of super class *Computer*. Now our factory classes will implement this interface and return their respective sub-class.

```

package com.journaldev.design.abstractfactory;
import com.journaldev.design.model.Computer;
import com.journaldev.design.model.PC;
public class PCFactory implements ComputerAbstractFactory {
 private String ram;
 private String hdd;
 private String cpu;
 public PCFactory(String ram, String hdd, String cpu){
 this.ram=ram;
 this.hdd=hdd;
 this.cpu=cpu;
 }
 @Override
 public Computer createComputer() {
 return new PC(ram,hdd,cpu);
 }
}

```

Similarly we will have a factory class for Server sub-class.

```

package com.journaldev.design.abstractfactory;
import com.journaldev.design.model.Computer;
import com.journaldev.design.model.Server;
public class ServerFactory implements ComputerAbstractFactory {
 private String ram;
 private String hdd;
 private String cpu;
 public ServerFactory(String ram, String hdd, String cpu){
 this.ram=ram;
 this.hdd=hdd;
 this.cpu=cpu;
 }
 @Override

```

```

public Computer createComputer() {
 return new Server(ram,hdd,cpu);
}
}
}

```

Now we will create a consumer class that will provide the entry point for the client classes to create sub-classes.

```

package com.journaldev.design.abstractfactory;
import com.journaldev.design.model.Computer;
public class ComputerFactory {
 public static Computer getComputer(ComputerAbstractFactory factory){
 return factory.createComputer();
 }
}

```

Notice that it's a simple class and `getComputer` method is accepting `ComputerAbstractFactory` argument and returning `Computer` object. At this point the implementation must be getting clear. Let's write a simple test method and see how to use the abstract factory to get the instance of sub-classes.

```

package com.journaldev.design.test;
import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;
public class TestDesignPatterns {
 public static void main(String[] args) {
 testAbstractFactory();
 }
 private static void testAbstractFactory() {
 Computer pc=com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new
 PCFactory("2 GB", "500 GB", "2.4 GHz"));
 Computer
 server=com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new
 ServerFactory("16 GB", "1 TB", "2.9 GHz"));
 System.out.println("AbstractFactory PC Config::"+pc);
 System.out.println("AbstractFactory Server Config::"+server);
 }
}

```

**Output of the above program will be:**

AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz

AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz

Here is the class diagram of abstract factory implementation.

### 1.3.3 Benefits of Abstract Factory Pattern

- Abstract Factory pattern provides approach to code for interface rather than implementation.
- Abstract Factory pattern is “factory of factories” and can be easily

extended to accommodate more products, for example we can add another sub-class Laptop and a factory LaptopFactory.

- Abstract Factory pattern is robust and avoid conditional logic of Factory pattern.

#### 1.3.4 Abstract Factory Pattern Examples in JDK

- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

### 4. Builder Pattern

Builder design pattern is a **creational design pattern** like **Factory Pattern** and **Abstract Factory Pattern**. This pattern was introduced to solve some of the problems with Factory and Abstract Factory design patterns when the Object contains a lot of attributes. There are three major issues with Factory and Abstract Factory design patterns when the Object contains a lot of attributes.

1. Too Many arguments to pass from client program to the Factory class that can be error prone because most of the time, the type of arguments are same and from client side it's hard to maintain the order of the argument.
2. Some of the parameters might be optional but in Factory pattern, we are forced to send all the parameters and optional parameters need to send as NULL.
3. If the object is heavy and its creation is complex, then all that complexity will be part of Factory classes that is confusing. We can solve the issues with large number of parameters by providing a constructor with required parameters and then different setter methods to set the optional parameters but the problem with this is that the Object state will be **inconsistent** until unless all the attributes are set explicitly. Builder pattern solves the issue with large number of optional parameters and inconsistent state by providing a way to build the object step-by-step and provide a method that will actually return the final Object.

#### 1.4.1 Builder Pattern Implementation

1. First of all you need to create a **static nested class** and then copy all the arguments from the outer class to the Builder class. We should follow the

naming convention and if the class name is *Computer* then builder class should be named as *ComputerBuilder*.

2. The Builder class should have a public constructor with all the required attributes as parameters.

3. Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.

4. The final step is to provide a *build()* method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Here is the sample code where we have a Computer class and ComputerBuilder class to build it.

```
package com.journaldev.design.builder;
public class Computer {
 //required parameters
 private String HDD;
 private String RAM;
 //optional parameters
 private boolean isGraphicsCardEnabled;
 private boolean isBluetoothEnabled;
 public String getHDD() {
 return HDD;
 }
 public String getRAM() {
 return RAM;
 }
 public boolean isGraphicsCardEnabled() {
 return isGraphicsCardEnabled;
 }
 public boolean isBluetoothEnabled() {
 return isBluetoothEnabled;
 }
 private Computer(ComputerBuilder builder) {
 this.HDD=builder.HDD;
 this.RAM=builder.RAM;
 this.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;
 this.isBluetoothEnabled=builder.isBluetoothEnabled;
 }
 //Builder Class
 public static class ComputerBuilder{
 // required parameters
 private String HDD;
 private String RAM;
 // optional parameters
 private boolean isGraphicsCardEnabled;
 private boolean isBluetoothEnabled;
 public ComputerBuilder(String hdd, String ram){
 this.HDD=hdd;
 this.RAM=ram;
 }
 public ComputerBuilder setGraphicsCardEnabled(Boolean isGraphicsCardEnabled) {

```

```

 this.isGraphicsCardEnabled = isGraphicsCardEnabled;
 return this;
 }
 public ComputerBuilder setBluetoothEnabled(Boolean isBluetoothEnabled) {
 this.isBluetoothEnabled = isBluetoothEnabled;
 return this;
 }
 public Computer build(){
 return new Computer(this);
 }
}
}

```

Notice that Computer class has only getter methods and no public constructor, so the only way to get a Computer object is through the ComputerBuilder class.

Here is a test program showing how to use Builder class to get the object.

```

package com.journaldev.design.test;
import com.journaldev.design.builder.Computer;
public class TestBuilderPattern {
 public static void main(String[] args) {
 //Using builder to get the object in a single line of code and
 //without any inconsistent state or arguments management issues
 Computer comp = new Computer.ComputerBuilder("500 GB", "2
 GB").setBluetoothEnabled(true).setGraphicsCardEnabled(true).build();
 }
}

```

#### 1.4.2 Builder Design Pattern Example in JDK

- `java.lang.StringBuilder#append()` (unsynchronized)
- `java.lang.StringBuffer#append()` (synchronized)

### 5. Prototype Pattern

**Prototype pattern** is one of the Creational Design pattern, so it provides a mechanism of object creation. Prototype pattern is used when the Object creation is a costly affair and requires a lot of time and resources and you have a similar object already existing. So this pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. This pattern uses java cloning to copy the object. It would be easy to understand this pattern with an example, suppose we have an Object that loads data from database. Now we need to modify this data in our program multiple times, so it's not a good idea to create the Object using `new` keyword and load all the data again from database. So the better approach is to clone the existing object into a new object and then do the data manipulation. Prototype design pattern mandates that the Object which you are copying should provide the copying feature. It should

not be done by any other class. However whether to use shallow or deep copy of the Object properties depends on the requirements and it's a design decision.

### 1.5.1 Implementation

Here is a sample program showing implementation of Prototype pattern.

```
package com.journaldev.design.prototype;
import java.util.ArrayList;
import java.util.List;
public class Employees implements Cloneable{
 private List<String> empList;
 public Employees(){
 empList = new ArrayList<String>();
 }
 public Employees(List<String> list){
 this.empList=list;
 }
 public void loadData(){
 //read all employees from database and put into the list
 empList.add("Pankaj");
 empList.add("Raj");
 empList.add("David");
 empList.add("Lisa");
 }
 public List<String> getEmpList() {
 return empList;
 }
 @Override
 public Object clone() throws CloneNotSupportedException{
 List<String> temp = new ArrayList<String>();
 for(String s : this.getEmpList()){
 temp.add(s);
 }
 return new Employees(temp);
 }
}
```

Notice that the clone method is overridden to provide a deep copy of the employees list. Here is the test program that will show the benefit of prototype pattern usage.

```
package com.journaldev.design.test;
import java.util.List;
import com.journaldev.design.prototype.Employees;
public class PrototypePatternTest {
 public static void main(String[] args) throws CloneNotSupportedException {
 Employees emps = new Employees();
 emps.loadData();
 //Use the clone method to get the Employee object
 Employees empsNew = (Employees) emps.clone();
```

```

Employees empsNew1 = (Employees) emps.clone();
List<String> list = empsNew.getEmpList();
list.add("John");
List<String> list1 = empsNew1.getEmpList();
list1.remove("Pankaj");
System.out.println("emps List: "+emps.getEmpList());
System.out.println("empsNew List: "+list);
System.out.println("empsNew1 List: "+list1);
}
}

```

**Output of the above program is:**

```

emps HashMap: [Pankaj, Raj, David, Lisa]
empsNew HashMap: [Pankaj, Raj, David, Lisa, John]
empsNew1 HashMap: [Raj, David, Lisa]

```

If the object cloning was not provided, every time we need to make database call to fetch the employee list and then do the manipulations that would have been resource and time consuming.

## Structural Design Patterns

Structural patterns provide different ways to create a class structure, for example using inheritance and composition to create a large object from small objects.

### 6. Adapter Design Pattern

**Adapter design pattern** is one of the **structural design pattern** and it's used so that two unrelated interfaces can work together. The object that joins these unrelated interface is called an **Adapter**. As a real life example, we can think of a mobile charger as an adapter because mobile battery needs 3 volts to charge but the normal socket produces either 120V (US) or 240V (India). So the mobile charger works as an adapter between mobile charging socket and the wall socket. We will try to implement multi-adapter using adapter design pattern in this tutorial. So first of all we will have two classes – Volt (to measure volts) and Socket (producing constant volts of 120V).

#### 1.6.1 Implementation

```

package com.journaldev.design.adapter;
public class Volt {
 private int volts;
 public Volt(int v){
 this.volts=v;
 }
}

```

```

 }
 public int getVolts() {
 return volts;
 }
 public void setVolts(int volts) {
 this.volts = volts;
 }
}

package com.journaldev.design.adapter;
public class Socket {
 public Volt getVolt(){
 return new Volt(120);
 }
}

```

Now we want to build an adapter that can produce 3 volts, 12 volts and default 120 volts. So first of all we will create an adapter interface with these methods.

```

package com.journaldev.design.adapter;
public interface SocketAdapter {
 public Volt get120Volt();
 public Volt get12Volt();
 public Volt get3Volt();
}

```

## 1.6.2 Two Way Adapter Pattern

While implementing Adapter pattern, there are two approaches – class adapter and object adapter, however both these approaches produce same result.

1. **Class Adapter** – This form uses **java inheritance** and extends the source interface, in our case Socket class.
2. **Object Adapter** – This form uses **Java Composition** and adapter contains the source object.

## 1.6.3 Class Adapter Implementation

Here is the **class adapter** approach implementation of our adapter.

```

package com.journaldev.design.adapter;
//Using inheritance for adapter pattern
public class SocketClassAdapterImpl extends Socket implements
SocketAdapter{
 @Override
 public Volt get120Volt() {
 return getVolt();
 }
}

```

```

@Override
public Volt get12Volt() {
 Volt v= getVolt();
 return convertVolt(v,10);
}
@Override
public Volt get3Volt() {
 Volt v= getVolt();
 return convertVolt(v,40);
}
private Volt convertVolt(Volt v, int i) {
 return new Volt(v.getVolts()/i);
}
}

```

## 1.6.4 Object Adapter Implementation

Here is the **Object adapter** implementation of our adapter.

```

package com.journaldev.design.adapter;
public class SocketObjectAdapterImpl implements SocketAdapter{
 //Using Composition for adapter pattern
 private Socket sock = new Socket();
 @Override
 public Volt get120Volt() {
 return sock.getVolt();
 }
 @Override
 public Volt get12Volt() {
 Volt v= sock.getVolt();
 return convertVolt(v,10);
 }
 @Override
 public Volt get3Volt() {
 Volt v= sock.getVolt();
 return convertVolt(v,40);
 }
 private Volt convertVolt(Volt v, int i) {
 return new Volt(v.getVolts()/i);
 }
}

```

Notice that both the adapter implementations are almost same and they implement the `SocketAdapter` interface. The adapter interface can also be an **abstract class**.

Here is a test program to consume our adapter implementation.

```

package com.journaldev.design.test;
import com.journaldev.design.adapter.SocketAdapter;
import com.journaldev.design.adapter.SocketClassAdapterImpl;
import com.journaldev.design.adapter.SocketObjectAdapterImpl;
import com.journaldev.design.adapter.Volt;
public class AdapterPatternTest {
 public static void main(String[] args) {
 testClassAdapter();
 }
}

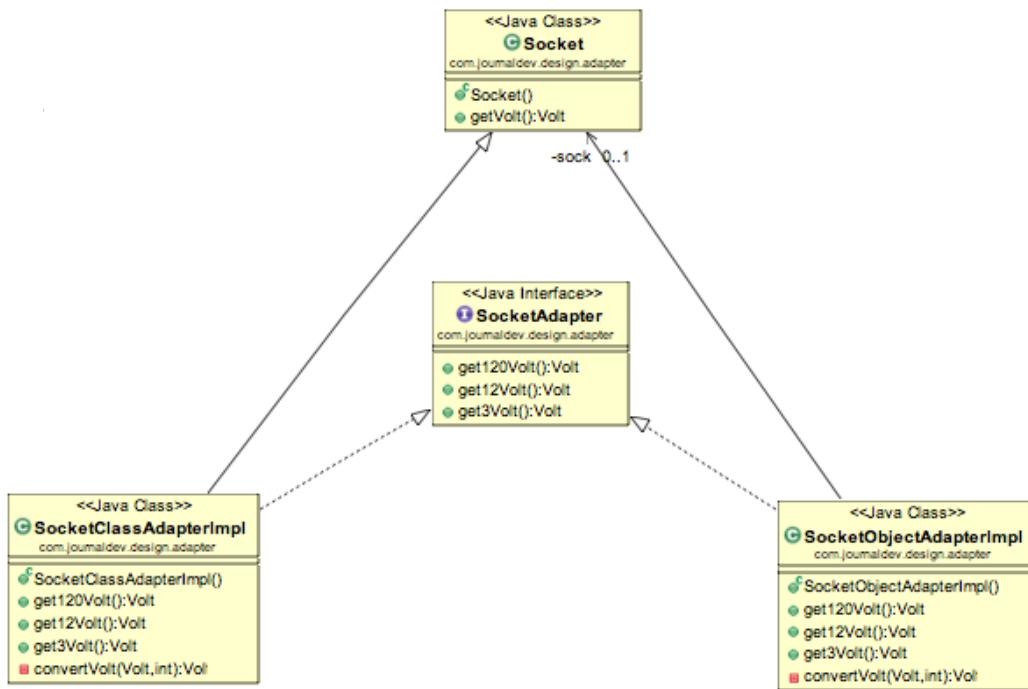
```

```
 testObjectAdapter();
 }
private static void testObjectAdapter() {
 SocketAdapter sockAdapter = new SocketObjectAdapterImpl();
 Volt v3 = getVolt(sockAdapter,3);
 Volt v12 = getVolt(sockAdapter,12);
 Volt v120 = getVolt(sockAdapter,120);
 System.out.println("v3 volts using Object Adapter=" + v3.getVolts());
 System.out.println("v12 volts using Object Adapter=" + v12.getVolts());
 System.out.println("v120 volts using Object Adapter=" + v120.getVolts());
}
private static void testClassAdapter() {
 SocketAdapter sockAdapter = new SocketClassAdapterImpl();
 Volt v3 = getVolt(sockAdapter,3);
 Volt v12 = getVolt(sockAdapter,12);
 Volt v120 = getVolt(sockAdapter,120);
 System.out.println("v3 volts using Class
Adapter=" + v3.getVolts());
 System.out.println("v12 volts using Class Adapter=" + v12.getVolts());
 System.out.println("v120 volts using Class Adapter=" + v120.getVolts());
}
private static Volt getVolt(SocketAdapter sockAdapter, int i) {
 switch (i){
 case 3: return sockAdapter.get3Volt();
 case 12: return sockAdapter.get12Volt();
 case 120: return sockAdapter.get120Volt();
 default: return sockAdapter.get120Volt();
 }
}
```

When we run above test program, we get following output.

v3 volts using Class Adapter=3  
v12 volts using Class Adapter=12  
v120 volts using Class Adapter=120  
v3 volts using Object Adapter=3  
v12 volts using Object Adapter=12  
v120 volts using Object Adapter=120

## 1.6.5 Adapter Pattern Class Diagram



## 1.6.6 Adapter Pattern Example in JDK

- `java.util.Arrays#asList()`
- `java.io.InputStreamReader(InputStream)` (returns a Reader)
- `java.io.OutputStreamWriter(OutputStream)` (returns a Writer)

### 7. Composite Pattern

**Composite pattern** is one of the **Structural design pattern** and is used when we have to represent a part-whole hierarchy. When we need to reate a structure in a way that the objects in the structure has to be treated the same way, we can apply composite design pattern.

Let's understand it with a real life example – A diagram is a structure that consists of Objects such as Circle, Lines, Triangle etc and when we fill the drawing with color (say Red), the same color also gets applied to the objects in the drawing. Here drawing is made up of different parts and they all have same operations. Composite Pattern consists of following objects.

#### 1.7.1 Components

1. **Base Component** – Base component is the interface for all objects in the composition, client program uses base component to work with the

objects in the composition. It can be an interface or an **abstract class** with some methods common to all the objects.

2. **Leaf** – Defines the behaviour for the elements in the composition. It is the building block for the composition and implements base component. It doesn't have references to other Components.

3. **Composite** – It consists of leaf elements and implements the operations in base component.

Here I am applying composite design pattern for the drawing scenario.

### 1.7.2Base Component

Base component defines the common methods for leaf and composites, we can create a *class Shape* with a method *draw(String fillColor)* to draw the shape with given color.

```
package com.journaldev.design.composite;
public interface Shape {
 public void draw(String fillColor);
}
```

### 1.7.3Leaf Objects

Leaf implements base component and these are the building block for the composite. We can create multiple leaf objects such as Triangle, Circle etc.

```
package com.journaldev.design.composite;
public class Triangle implements Shape {
 @Override
 public void draw(String fillColor) {
 System.out.println("Drawing Triangle with color "+fillColor);
 }
}
package com.journaldev.design.composite;
public class Circle implements Shape {
 @Override
 public void draw(String fillColor) {
 System.out.println("Drawing Circle with color "+fillColor);
 }
}
```

### 1.7.4Composite

A composite object contains group of leaf objects and we should provide some helper methods to add or delete leafs from the group. We can also provide a method to remove all the elements from the group.

```
package com.journaldev.design.composite;
import java.util.ArrayList;
import java.util.List;
public class Drawing implements Shape{
 //collection of Shapes
 private List<Shape> shapes = new ArrayList<Shape>();
 @Override
 public void draw(String fillColor) {
```

```

 for(Shape sh : shapes){
 sh.draw(fillColor);
 }
 }
 //adding shape to drawing
 public void add(Shape s){
 this.shapes.add(s);
 }
 //removing shape from drawing
 public void remove(Shape s){
 shapes.remove(s);
 }
 //removing all the shapes
 public void clear(){
 System.out.println("Clearing all the shapes from drawing");
 this.shapes.clear();
 }
}

```

Notice that composite also implements component and behaves similar to leaf except that it can contain group of leaf elements.

Our composite pattern implementation is ready and we can test it with a client program.

```

package com.journaldev.design.test;
import com.journaldev.design.composite.Circle;
import com.journaldev.design.composite.Drawing;
import com.journaldev.design.composite.Shape;
import com.journaldev.design.composite.Triangle;
public class TestCompositePattern {
 public static void main(String[] args) {
 Shape tri = new Triangle();
 Shape tri1 = new Triangle();
 Shape cir = new Circle();
 Drawing drawing = new Drawing();
 drawing.add(tri1);
 drawing.add(tri1);
 drawing.add(cir);
 drawing.draw("Red");
 drawing.clear();
 drawing.add(tri);
 drawing.add(cir);
 drawing.draw("Green");
 }
}

```

**Output of the above program is:**

```

Drawing Triangle with color Red
Drawing Triangle with color Red
Drawing Circle with color Red
Clearing all the shapes from drawing
Drawing Triangle with color Green
Drawing Circle with color Green

```

### 1.7.5 Important Points about Composite Pattern

- Composite pattern should be applied only when the group of objects

should behave as the single object.

□ Composite pattern can be used to create a tree like structure.

`java.awt.Container#add(Component)` is a great example of Composite pattern in java and used a lot in Swing.

## 8. Proxy Pattern

**Proxy Design pattern** is one of the **Structural design pattern** and in my opinion one of the simplest pattern to understand. Proxy pattern intent according to GoF is:

The definition itself is very clear and proxy pattern is used when we want to provide controlled access of a functionality. Let's say we have a class that can run some command on the system. Now if we are using it, its fine but if we want to give this program to a client application, it can have severe issues because client program can issue command to delete some system files or change some settings that you don't want. Here a proxy class can be created to provide controlled access of the program.

“Provide a surrogate or placeholder for another object to control access to it”

### 1.8.1 Main Class

Since we code Java in terms of interfaces, here is our interface and its implementation class.

```
package com.journaldev.design.proxy;
public interface CommandExecutor {
 public void runCommand(String cmd) throws Exception;
}

package com.journaldev.design.proxy;
import java.io.IOException;
public class CommandExecutorImpl implements CommandExecutor {
 @Override
 public void runCommand(String cmd) throws IOException {
 //some heavy implementation
 Runtime.getRuntime().exec(cmd);
 System.out.println("'" + cmd + "' command executed.");
 }
}
```

### 1.8.2 Proxy Class

Now we want to provide only admin users to have full access of above class, if the user is not admin then only limited commands will be allowed. Here is our very simple proxy class implementation.

```

package com.journaldev.design.proxy;
public class CommandExecutorProxy implements CommandExecutor {
 private boolean isAdmin;
 private CommandExecutor executor;
 public CommandExecutorProxy(String user, String pwd){
 if("Pankaj".equals(user) && "J@urnald$v".equals(pwd))
 isAdmin=true;
 executor = new CommandExecutorImpl();
 }
 @Override
 public void runCommand(String cmd) throws Exception {
 if(isAdmin){
 executor.runCommand(cmd);
 }else{
 if(cmd.trim().startsWith("rm")){
 throw new Exception("rm command is not allowed for nonadmin
users.");
 }else{
 executor.runCommand(cmd);
 }
 }
 }
}

```

### 1.8.3 Proxy Pattern Client Test Program

```

package com.journaldev.design.test;
import com.journaldev.design.proxy.CommandExecutor;
import com.journaldev.design.proxy.CommandExecutorProxy;
public class ProxyPatternTest {
 public static void main(String[] args){
 CommandExecutor executor = new CommandExecutorProxy("Pankaj","wrong_pwd");
 try {
 executor.runCommand("ls -ltr");
 executor.runCommand(" rm -rf abc.pdf");
 } catch (Exception e) {
 System.out.println("Exception Message::"+e.getMessage());
 }
 }
}

```

Output of above test program is:

'ls -ltr' command executed.

Exception **Message::**rm command is not allowed **for** non-admin users.

Proxy pattern common uses are to control access or to provide a wrapper implementation for better performance. Java RMI whole package uses proxy pattern.

### 9. Flyweight Pattern

According to GoF, **flyweight design pattern** intent is:

Flyweight design pattern is a **Structural design pattern** like **Facade pattern**, **Adapter Pattern** and **Decorator pattern**. Flyweight design pattern is used when we need to create a lot of Objects of a class. Since every object

consumes memory space that can be crucial for low memory devices, such as mobile devices or embedded systems, flyweight design pattern can be applied to reduce the load on memory by sharing objects. Before we apply flyweight design pattern, we need to consider following factors:

- The number of Objects to be created by application should be huge.
- The object creation is heavy on memory and it can be time consuming too.
- The object properties can be divided into intrinsic and extrinsic properties, extrinsic properties of an Object should be defined by the client program. To apply flyweight pattern, we need to divide Object property into **intrinsic** and **extrinsic** properties. Intrinsic properties make the Object unique whereas extrinsic properties are set by client code and used to perform different operations. For example, an Object Circle can have extrinsic properties such as color and width.

For applying flyweight pattern, we need to create a **Flyweight factory** that returns the shared objects. For our example, let's say we need to create a drawing with lines and Ovals. So we will have an interface Shape and its concrete implementations as *Line* and *Oval*. Oval class will have intrinsic property to determine whether to fill the Oval with given color or not whereas Line will not have any intrinsic property.

“Use sharing to support large numbers of fine-grained objects efficiently”

### 1.9.1 Flyweight Interface and Concrete Classes

```
/*
 *
 */
package com.game.structural;

import java.util.HashMap;

interface ShapeFlyweight {
 void draw();
}

class CircleFlyweight implements ShapeFlyweight {
 private int x;
 private int y;
 private String color;
 private int radius;

 CircleFlyweight(String color) {
 setColor(color);
 System.out.println("Created new circle with Color:" + getColor());
 }

 @Override
 public void draw() {
```

```

 System.out.println(
 "Drawn circle with x:" + getX() + " y:" + y + " Radius:" + getRadius()
+ " color:" + getColor());
}

public int getX() {
 return x;
}

public void setX(int x) {
 this.x = x;
}

public int getY() {
 return y;
}

public void setY(int y) {
 this.y = y;
}

public String getColor() {
 return color;
}

public void setColor(String color) {
 this.color = color;
}

public int getRadius() {
 return radius;
}

public void setRadius(int radius) {
 this.radius = radius;
}

}

class FactoryCircle {
 private static HashMap circleMap = new HashMap();

 public static CircleFlyweight getCircle(String color) {
 CircleFlyweight circle = (CircleFlyweight) circleMap.get(color);
 if (circle == null) {
 circle = new CircleFlyweight(color);
 circleMap.put(color, circle);
 }
 return circle;
 }

 /**
 * @author bittu
 */
}

```

```

public class Flyweight {
 private static String []colors = {"Red","Green","Blue","Black","White"};
 /**
 *
 */
 public Flyweight() {
 // TODO Auto-generated constructor stub
 }

 /**
 * @param args
 */
 public static void main(String[] args) {
 for(int i = 0; i<20;i++) {
 CircleFlyweight circle = FactoryCircle.getCircle(getRandomColor());
 circle.setX(getRandomX());
 circle.setY(getRandomY());
 circle.setRadius(100);
 circle.draw();
 }
 }

 private static int getRandomX() {
 // TODO Auto-generated method stub
 return (int) Math.random()*100;
 }
 private static int getRandomY() {
 // TODO Auto-generated method stub
 return (int) Math.random()*100;
 }

 private static String getRandomColor() {
 return colors[(int) (Math.random()*colors.length)];
 }
}

```

Notice that I have intentionally introduced delay in creating the Object of concrete classes to make the point that flyweight pattern can be used for Objects that takes a lot of time while instantiated.

### 1.9.2 Flyweight Pattern Example in JDK

All the [wrapper classes](#) `valueOf()` method uses cached objects showing use of Flyweight design pattern. The best example is [Java String](#) class [String Pool](#) implementation.

### 1.9.3 Important Points

- In our example, the client code is not forced to create object using Flyweight factory but we can force that to make sure client code uses flyweight pattern implementation but its a complete design decision for particular application.

- Flyweight pattern introduces complexity and if number of shared objects are huge then there is a trade off between memory and time, so we need to use it judiciously based on our requirements.
- Flyweight pattern implementation is not useful when the number of intrinsic properties of Object is huge, making implementation of Factory class complex.

## 10. Facade Pattern

**Facade Pattern** is one of the **Structural design patterns** (such as [Adapter pattern](#) and [Decorator pattern](#)) and used to help client applications to easily interact with the system. According to GoF Facade design pattern is:

Provide a unified interface to a set of interfaces in a subsystem. Facade Pattern defines a higher-level interface that makes the subsystem easier to use.

Suppose we have an application with set of interfaces to use MySql/Oracle database and to generate different types of reports, such as HTML report, PDF report etc. So we will have different set of interfaces to work with different types of database. Now a client application can use these interfaces

to get the required database connection and generate reports. But when the

complexity increases or the interface behavior names are confusing, client application will find it difficult to manage it. So we can apply Facade pattern here and provide a [wrapper](#) interface on top of the existing interface to help client application.

### 1.10.1 Set of Interfaces

We can have two helper interfaces, namely MySqlHelper and OracleHelper.

```
package com.journaldev.design.facade;
import java.sql.Connection;
public class MySqlHelper {
 public static Connection getMySqlDBConnection(){
 //get MySql DB connection using connection parameters
 return null;
 }
 public void generateMySqlPDFReport(String tableName, Connection con){
 //get data from table and generate pdf report
 }
 public void generateMySqlHTMLReport(String tableName, Connection con){
 //get data from table and generate pdf report
 }
}
```

```

package com.journaldev.design.facade;
import java.sql.Connection;
public class OracleHelper {
 public static Connection getOracleDBConnection(){
 //get MySql DB connection using connection parameters
 return null;
 }
 public void generateOraclePDFReport(String tableName, Connection con){
 //get data from table and generate pdf report
 }
 public void generateOracleHTMLReport(String tableName, Connection con){
 //get data from table and generate pdf report
 }
}

```

## 1.10.2 Facade Interface

We can create a Facade interface like below. Notice the use of Java Enum for type safety.

```

package com.journaldev.design.facade;
import java.sql.Connection;
public class HelperFacade {
 public static void generateReport(DBTypes dbType, ReportTypes reportType, String
tableName){
 Connection con = null;
 switch (dbType){
 case MYSQL:
 con = MySqlHelper.getMySqlDBConnection();
 MySqlHelper mySqlHelper = new MySqlHelper();
 switch(reportType){
 case HTML:
 mySqlHelper.generateMySqlHTMLReport(tableName, con);
 break;
 case PDF:
 mySqlHelper.generateMySqlPDFReport(tableName, con);
 break;
 }
 break;
 case ORACLE:
 con = OracleHelper.getOracleDBConnection();
 OracleHelper oracleHelper = new OracleHelper();
 switch(reportType){
 case HTML:
 oracleHelper.generateOracleHTMLReport(tableName, con);
 break;
 case PDF:
 oracleHelper.generateOraclePDFReport(tableName, con);
 break;
 }
 break;
 }
 }
 public static enum DBTypes{
 MYSQL,ORACLE;
 }
 public static enum ReportTypes{

```

```
HTML,PDF;
}
}
```

### 1.10.3 Client Program

Now let's see client code without using Facade and using Facade interface.

```
package com.journaldev.design.test;
import java.sql.Connection;
import com.journaldev.design.facade.HelperFacade;
import com.journaldev.design.facade.MySqlHelper;
import com.journaldev.design.facade.OracleHelper;
public class FacadePatternTest {
public static void main(String[] args) {
 String tableName="Employee";
 //generating MySql HTML report and Oracle PDF report without using Facade
 Connection con = MySqlHelper.getMySqlDBConnection();
 MySqlHelper mySqlHelper = new MySqlHelper();
 mySqlHelper.generateMySqlHTMLReport(tableName, con);
 Connection con1 = OracleHelper.getOracleDBConnection();
 OracleHelper oracleHelper = new OracleHelper();
 oracleHelper.generateOraclePDFReport(tableName, con1);
 //generating MySql HTML report and Oracle PDF report using Facade
 HelperFacade.generateReport(HelperFacade.DBTypes.MYSQL,
 HelperFacade.ReportTypes.HTML, tableName);
 HelperFacade.generateReport(HelperFacade.DBTypes.ORACLE,
 HelperFacade.ReportTypes.PDF, tableName);
}
}
```

As you can see that using Facade interface is a lot easier and cleaner way and avoid having a lot of logic at client side. JDBC Driver Manager Class to get the database connection is a wonderful example of facade pattern.

### 1.10.4 Important Points

- Facade pattern is more like a helper for client applications, it doesn't hide subsystem interfaces from the client. Whether to use Facade or not is completely dependent on client code.
- Facade pattern can be applied at any point of development, usually when the number of interfaces grow and system gets complex.
- Subsystem interfaces are not aware of Facade and they shouldn't have any reference of the Facade interface.
- Facade pattern should be applied for similar kind of interfaces, its purpose is to provide a single interface rather than multiple interfaces that does the similar kind of jobs.
- We can use **Factory pattern** with Facade to provide better interface to client systems.

## 11. Bridge Pattern

When we have interface hierarchies in both interfaces as well as implementations, then **builder design pattern** is used to decouple the interfaces from implementation and hiding the implementation details from the client programs. Like [Adapter pattern](#), its one of the **Structural design pattern**.

According to GoF bridge design pattern is:

The implementation of bridge design pattern follows the notion to prefer **Composition over inheritance**.

If we look into this design pattern with example, it will be easy to understand. Let's say we have an interface hierarchy in both interfaces and implementations like below image.

**“Decouple an abstraction from its implementation so that the two can vary independently”**

Now we will use bridge design pattern to decouple the interfaces from implementation and the UML diagram for the classes and interfaces after applying bridge pattern will look like below image.

Notice the bridge between *Shape* and *Color* interfaces and use of composition in implementing the bridge pattern.

### 1.11.1 Implementation

Here is the java code for Shape and Color interfaces.

```
package com.journaldev.design.bridge;
public interface Color {
 public void applyColor();
}

package com.journaldev.design.bridge;
public abstract class Shape {
 //Composition - implementor
 protected Color color;
 //constructor with implementor as input argument
 public Shape(Color c){
 this.color=c;
 }
 abstract public void applyColor();
}
}
```

We have Triangle and Pentagon implementation classes as below.

```
package com.journaldev.design.bridge;
public class Triangle extends Shape{
 public Triangle(Color c) {
 super(c);
 }
 @Override
 public void applyColor() {
 System.out.print("Triangle filled with color ");
 color.applyColor();
 }
}
```

```

 }
}

package com.journaldev.design.bridge;
public class Pentagon extends Shape{
 public Pentagon(Color c) {
 super(c);
 }
 @Override
 public void applyColor() {
 System.out.print("Pentagon filled with color ");
 color.applyColor();
 }
}

```

Here are the implementation classes for RedColor and GreenColor.

```

package com.journaldev.design.bridge;
public class RedColor implements Color{
 public void applyColor(){
 System.out.println("red.");
 }
}

package com.journaldev.design.bridge;
public class GreenColor implements Color{
 public void applyColor(){
 System.out.println("green.");
 }
}

```

Let's test our bridge pattern implementation with a test program.

```

package com.journaldev.design.test;
import com.journaldev.design.bridge.GreenColor;
import com.journaldev.design.bridge.Pentagon;
import com.journaldev.design.bridge.RedColor;
import com.journaldev.design.bridge.Shape;
import com.journaldev.design.bridge.Triangle;
public class BridgePatternTest {
 public static void main(String[] args) {
 Shape tri = new Triangle(new RedColor());
 tri.applyColor();
 Shape pent = new Pentagon(new GreenColor());
 pent.applyColor();
 }
}

```

Output of above class is:

Triangle filled with color red.

Pentagon filled with color green.

### 1.11.2 Importance

Bridge design pattern can be used when both abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

## 12. Decorator Pattern

**Decorator design pattern** is used to modify the functionality of an object at runtime. At the same time other instances of the same class will not be affected by this, so individual object gets the modified behavior. Decorator design pattern is one of the structural design pattern (such as [Adapter Pattern](#), [Bridge Pattern](#), [Composite Pattern](#)) and uses abstract classes or interface with [composition](#) to implement. We use [inheritance](#) or composition to extend the behavior of an object but this is done at compile time and its applicable to all the instances of the class. We can't add any new functionality or remove any existing behavior at runtime – this is when Decorator pattern comes into picture. Suppose we want to implement different kinds of cars – we can create interface Car to define the assemble method and then we can have a Basic car, further more we can extend it to Sports car and Luxury Car. The implementation hierarchy will look like below image.

But if we want to get a car at runtime that has both the features of sports car and luxury car, then the implementation gets complex and if further more we want to specify which features should be added first, it gets even more complex. Now image if we have ten different kind of cars, the implementation logic using inheritance and composition will be impossible to manage. To solve this kind of programming situation, we apply decorator pattern.

We need to have following types to implement decorator design pattern.

### 1.12.1 Component Interface

The interface or [abstract class](#) defining the methods that will be implemented. In our case Car will be the component interface.

```
package com.journaldev.design.decorator;
public interface Car {
 public void assemble();
}
```

### 1.12.2 Component Implementation

The basic implementation of the component interface. We can have BasicCar class as our component implementation.

```
package com.journaldev.design.decorator;
public class BasicCar implements Car {
 @Override
 public void assemble() {
 System.out.print("Basic Car.");
 }
}
```

### 1.12.3 Decorator

Decorator class implements the component interface and it has a HAS-A relationship with the component interface. The component variable should be accessible to the child decorator classes, so we will make this variable protected.

```
package com.journaldev.design.decorator;
public class CarDecorator implements Car {
 protected Car car;
 public CarDecorator(Car c){
 this.car=c;
 }
 @Override
 public void assemble() {
 this.car.assemble();
 }
}
```

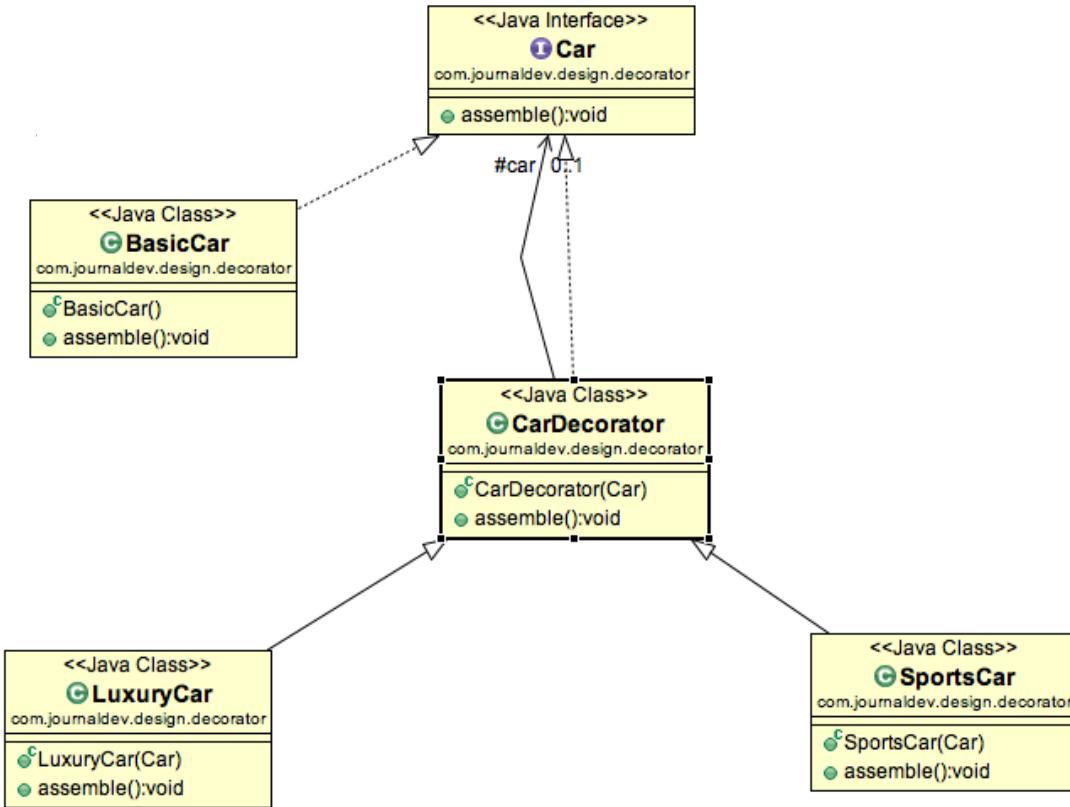
### 1.12.4 Concrete Decorators

Extending the base decorator functionality and modifying the component behavior accordingly. We can have concrete decorator classes as LuxuryCar and SportsCar.

```
package com.journaldev.design.decorator;
public class SportsCar extends CarDecorator {
 public SportsCar(Car c) {
 super(c);
 }
 @Override
 public void assemble(){
 car.assemble();
 System.out.print(" Adding features of Sports Car.");
 }
}

package com.journaldev.design.decorator;
public class LuxuryCar extends CarDecorator {
 public LuxuryCar(Car c) {
 super(c);
 }
 @Override
 public void assemble(){
 car.assemble();
 System.out.print(" Adding features of Luxury Car.");
 }
}
```

## 1.12.5 Decorator Pattern Class Diagram



## 1.12.6 Decorator Pattern Client Program

```

package com.journaldev.design.test;
import com.journaldev.design.decorator.BasicCar;
import com.journaldev.design.decorator.Car;
import com.journaldev.design.decorator.LuxuryCar;
import com.journaldev.design.decorator.SportsCar;
public class DecoratorPatternTest {
 public static void main(String[] args) {
 Car sportsCar = new SportsCar(new BasicCar());
 sportsCar.assemble();
 System.out.println("\n*****");
 Car sportsLuxuryCar = new SportsCar(new LuxuryCar(new BasicCar()));
 sportsLuxuryCar.assemble();
 }
}

```

Notice that client program can create different kinds of Object at runtime and they can specify the order of execution too.

Output of above test program is:

Basic Car. Adding features of Sports Car.

\*\*\*\*\*

Basic Car. Adding features of Luxury Car. Adding features of Sports Car.

### 1.12.7 Important Points

- Decorator pattern is helpful in providing runtime modification abilities and hence more flexible. It's easy to maintain and extend when the number of choices are more.
- The disadvantage of decorator pattern is that it uses a lot of similar kind of objects (decorators).
- Decorator pattern is used a lot in [Java IO classes](#), such as [FileReader](#), [BufferedReader](#) etc.

## **Behavioral Design Patterns**

Behavioral patterns provide solution for the better interaction between objects and how to provide loose coupling and flexibility to extend easily.

### 13. Template Method Pattern

**Template Method** is a **behavioral design pattern** and it's used to create a method stub and deferring some of the steps of implementation to the subclasses. **Template method** defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses. Let's understand this pattern with an example, suppose we want to provide an algorithm to build a house. The steps need to be performed to build a house are – building foundation, building pillars, building walls and windows. The important point is that we can't change the order of execution because we can't build windows before building the foundation. So in this case we can create a template method that will use different methods to build the house.

Now building the foundation for a house is same for all type of houses, whether it's a wooden house or a glass house. So we can provide base implementation for this, if subclasses want to override this method, they can but mostly it's common for all the types of houses.

To make sure that subclasses don't override the template method, we should make it final.

### 1.13.1 Template Method Abstract Class

Since we want some of the methods to be implemented by subclasses, we have to make our base class as [abstract class](#).

```
package com.journaldev.design.template;
public abstract class HouseTemplate {
 //template method, final so subclasses can't override
 public final void buildHouse(){}
```

```

 buildFoundation();
 buildPillars();
 buildWalls();
 buildWindows();
 System.out.println("House is built.");
 }
 //default implementation
 private void buildWindows() {
 System.out.println("Building Glass Windows");
 }
 //methods to be implemented by subclasses
 public abstract void buildWalls();
 public abstract void buildPillars();
 private void buildFoundation() {
 System.out.println("Building foundation with cement,iron rods and sand");
 }
}

```

*buildHouse()* is the template method and defines the order of execution for performing several steps.

### 1.13.2 Template Method Concrete Classes

We can have different type of houses, such as Wooden House and Glass House.

```

package com.journaldev.design.template;
public class WoodenHouse extends HouseTemplate {
 @Override
 public void buildWalls() {
 System.out.println("Building Wooden Walls");
 }
 @Override
 public void buildPillars() {
 System.out.println("Building Pillars with Wood coating");
 }
}

```

We could have overridden other methods also, but for simplicity I am not doing that.

```

package com.journaldev.design.template;
public class GlassHouse extends HouseTemplate {
 @Override
 public void buildWalls() {
 System.out.println("Building Glass Walls");
 }
 @Override
 public void buildPillars() {
 System.out.println("Building Pillars with glass coating");
 }
}

```

### 1.13.3 Template Method Pattern Client

Let's test our template method pattern example with a test program.

```

package com.journaldev.design.template;
public class HousingClient {
 public static void main(String[] args) {

```

```

HouseTemplate houseType = new WoodenHouse();
//using template method
houseType.buildHouse();
System.out.println("*****");
houseType = new GlassHouse();
houseType.buildHouse();
}
}
}

```

Notice that client is invoking the template method of base class and depending of implementation of different steps, it's using some of the methods from base class and some of them from subclass.

**Output of the above program is:**

```

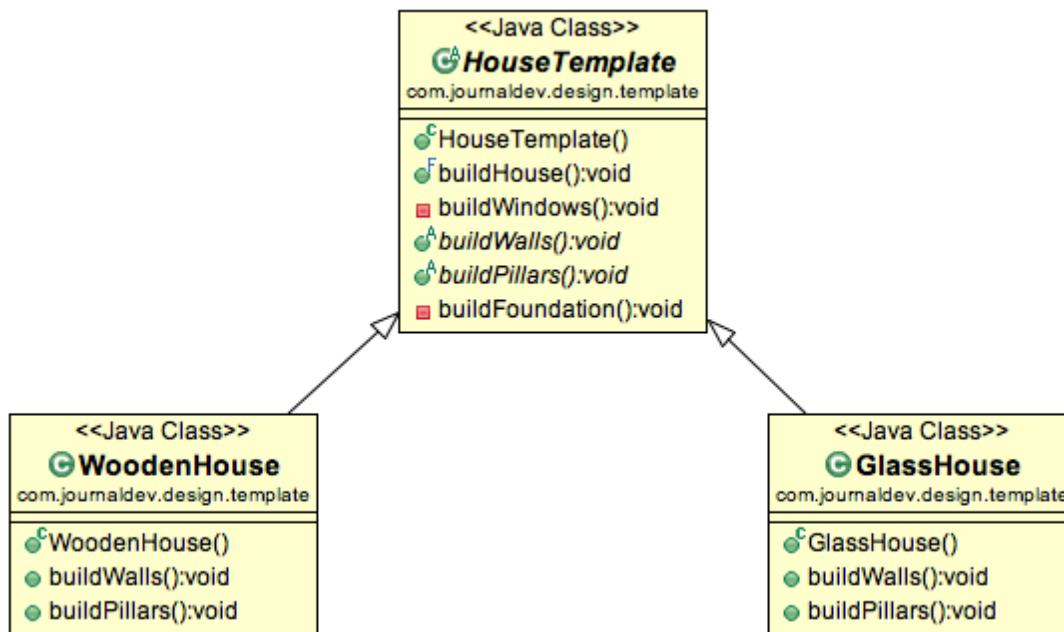
Building foundation with cement,iron rods and sand
Building Pillars with Wood coating
Building Wooden Walls
Building Glass Windows
House is built.

```

```

Building foundation with cement,iron rods and sand
Building Pillars with glass coating
Building Glass Walls
Building Glass Windows
House is built.
```

#### 1.13.4 Template Method Class Diagram



#### 1.13.5 Template Method Pattern in JDK

- All non-abstract methods of `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer`.
- All non-abstract methods of `java.util.AbstractList`,

`java.util.AbstractSet` and `java.util.AbstractMap`.

### 1.13.6 Important Points

- Template method should consists of certain steps whose order is fixed and for some of the methods, implementation differs from base class to subclass. Template method should be final.
- Most of the times, subclasses calls methods from super class but in template pattern, superclass template method calls methods from subclasses, this is known as **Hollywood Principle** – “don’t call us, we’ll call you”.
- Methods in base class with default implementation are referred as **Hooks** and they are intended to be overridden by subclasses, if you want some of the methods to be not overridden, you can make them final, for example in our case we can make `buildFoundation()` method final because if we don’t want subclasses to override it.

## 14. Mediator Pattern

**Mediator Pattern** is one of the **behavioral design pattern**, so it deals with the behaviors of objects. Mediator design pattern is used to provide a centralized communication medium between different objects in a system. According to GoF, mediator pattern intent is: Mediator design pattern is very helpful in an enterprise application where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly-coupled with each other that makes maintainability cost higher and not flexible to extend easily. Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose-coupling between objects. Air traffic controller is a great example of mediator pattern where the airport control room works as a mediator for communication between different flights. Mediator works as a router between objects and it can have its own logic to provide way of communication.

The system objects that communicate each other are called Colleagues. Usually we have an **interface or abstract class** that provides the contract for communication and then we have concrete implementation of mediators. For our example, we will try to implement a chat application where users can do group chat. Every user will be identified by its name and they can send and receive messages. The message sent by any user should be received by all the other users in the group.

**“Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another”**

### 1.14.1 Mediator Interface

First of all we will create Mediator interface that will define the contract for concrete mediators.

```
package com.journaldev.design.mediator;
public interface ChatMediator {
 public void sendMessage(String msg, User user);
 void addUser(User user);
}
```

### 1.14.2 Colleague Interface

Users can send and receive messages, so we can have User interface or abstract class. I am creating User as abstract class like below.

```
package com.journaldev.design.mediator;
public abstract class User {
 protected ChatMediator mediator;
 protected String name;
 public User(ChatMediator med, String name){
 this.mediator=med;
 this.name=name;
 }
 public abstract void send(String msg);
 public abstract void receive(String msg);
}
```

Notice that User has a reference to the mediator object, it's required for the communication between different users.

### 1.14.3 Concrete Mediator

Now we will create concrete mediator class, it will have a list of users in the group and provide logic for the communication between the users.

```
package com.journaldev.design.mediator;
import java.util.ArrayList;
import java.util.List;
public class ChatMediatorImpl implements ChatMediator {
 private List<User> users;
 public ChatMediatorImpl(){
 this.users=new ArrayList<>();
 }
 @Override
 public void addUser(User user){
 this.users.add(user);
 }
 @Override
```

```

public void sendMessage(String msg, User user) {
 for(User u : this.users){
 //message should not be received by the user sending it
 if(u != user){
 u.receive(msg);
 }
 }
}

```

#### 1.14.4 Concrete Colleague

Now we can create concrete User classes to be used by client system.

```

package com.journaldev.design.mediator;
public class UserImpl extends User {
 public UserImpl(ChatMediator med, String name) {
 super(med, name);
 }
 @Override
 public void send(String msg){
 System.out.println(this.name+": Sending Message="+msg);
 mediator.sendMessage(msg, this);
 }
 @Override
 public void receive(String msg) {
 System.out.println(this.name+": Received Message:"+msg);
 }
}

```

Notice that send() method is using mediator to send the message to the users and it has no idea how it will be handled by the mediator.

#### 1.14.5 Mediator Pattern Client

Let's test this our chat application with a simple program where we will create mediator and add users to the group and one of the user will send a message.

```

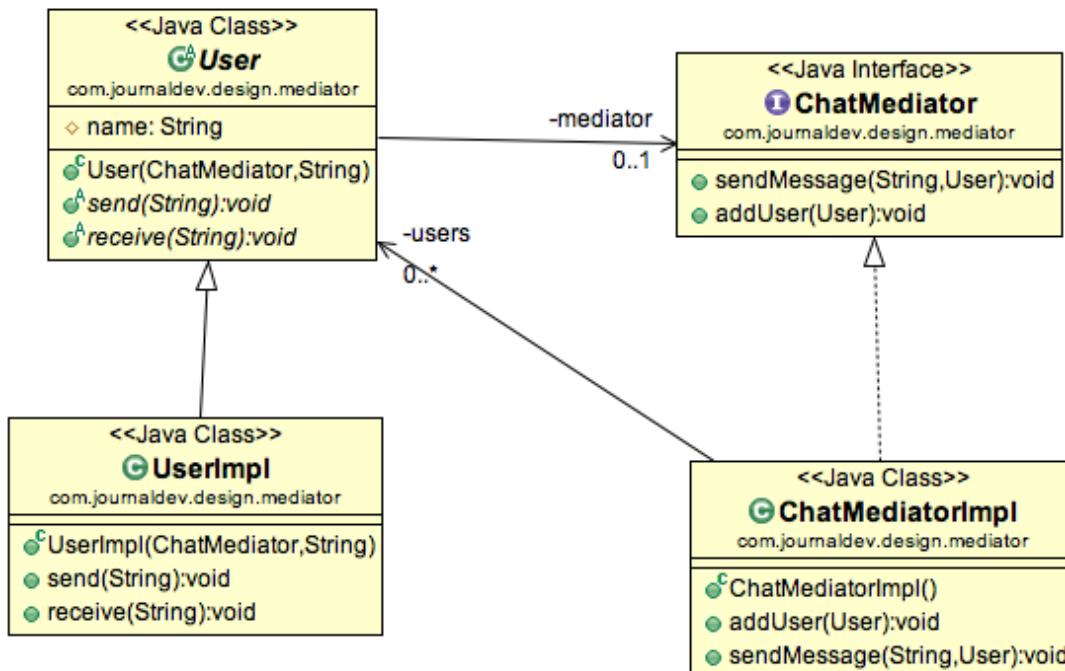
package com.journaldev.design.mediator;
public class ChatClient {
 public static void main(String[] args) {
 ChatMediator mediator = new ChatMediatorImpl();
 User user1 = new UserImpl(mediator, "Pankaj");
 User user2 = new UserImpl(mediator, "Lisa");
 User user3 = new UserImpl(mediator, "Saurabh");
 User user4 = new UserImpl(mediator, "David");
 mediator.addUser(user1);
 mediator.addUser(user2);
 mediator.addUser(user3);
 mediator.addUser(user4);
 user1.send("Hi All");
 }
}

```

Notice that client program is very simple and it has no idea how the message is getting handled and if mediator is getting user or not.  
Output of the above program is:

Pankaj: Sending Message=Hi All  
 Lisa: Received Message:Hi All  
 Saurabh: Received Message:Hi All  
 David: Received Message:Hi All

### 1.14.6 Mediator Pattern Class Diagram



### 1.14.7 Mediator Pattern in JDK

`java.util.Timer` class `scheduleXXX()` methods  
`Java Concurrency Executor` `execute()` method.  
`java.lang.reflect.Method` `invoke()` method.

### 1.14.8 Important Points

- Mediator pattern is useful when the communication logic between objects is complex, we can have a central point of communication that takes care of communication logic.
- Java Message Service (JMS) uses Mediator pattern along with **Observer pattern** to allow applications to subscribe and publish data to other applications.

- We should not use mediator pattern just to achieve loose-coupling because if the number of mediators will grow, then it will become hard to maintain them.

# Data Structures

Note: Considering the importance for coding by own, the links are to the websites for the appropriate solution to the problem.

Recommended: Please solve it on “**PRACTICE**” first, before moving on to the solution. **Do it on Write Board and then on IDE.**

## Notes:

- 2 Pointers solution is applicable mostly for sorted arrays
- Inorder to divide the problem(array) into subproblems – do take gcd()

## 1. Arrays

### 1. Find a pair in an array of size 'n', whose sum is X

Write a program that, given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x.

#### Examples:

**Input:** arr[] = {0, -1, 2, -3, 1}  
sum = -2

**Output:** -3, 1  
If we calculate the sum of the output,  
 $1 + (-3) = -2$

**Input:** arr[] = {1, -2, 1, 0, 5}  
sum = 0

**Output:** -1  
No valid pair exists.

## 2. Find a majority element in an array of size 'n'

Write a function which takes an array and prints the majority element (if it exists), otherwise prints “No Majority Element”. A **majority element** in an array  $A[]$  of size  $n$  is an element that appears more than  $n/2$  times (and hence there is at most one such element).

**Examples :**

**Input :** {3, 3, 4, 2, 4, 4, 2, 4, 4}

**Output :** 4

**Explanation:** The frequency of 4 is 5 which is greater than the half of the size of the array size.

**Input :** {3, 3, 4, 2, 4, 4, 2, 4}

**Output :** No Majority Element

**Explanation:** There is no element whose frequency is greater than the half of the size of the array size.

## 3. Find the number occurring odd number of times in a given array of size 'n'

Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in  $O(n)$  time & constant space.

**Examples :**

**Input :** arr = {1, 2, 3, 2, 3, 1, 3}

**Output :** 3

**Input :** arr = {5, 7, 2, 7, 5, 2, 5}

**Output :** 5

## 4. Algorithm to reverse an array

Given an array (or string), the task is to reverse the array/string.

**Examples :**

**Input :** arr[] = {1, 2, 3}

**Output :** arr[] = {3, 2, 1}

Input : arr[] = {4, 5, 1, 2}

Output : arr[] = {2, 1, 5, 4}

## 5. Algorithm to rotate array of size 'n' by 'd' elements

Write a function rotate(ar[], d, n) that rotates arr[] of size n by d elements.

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|

Rotation of the above array by 2 will make array

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 7 | 1 | 2 |
|---|---|---|---|---|---|---|

## 6. Algorithm to segregate 0's and 1's in an array

You are given an array of 0s and 1s in random order. Segregate 0s on left side and 1s on right side of the array. Traverse array only once.

Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]

Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1]

**7. Find the maximum difference between two elements such that larger element appears after the smaller element**

You are given an array of 0s and 1s in random order. Segregate 0s on left side and 1s on right side of the array. Traverse array only once.

Input array = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]

Output array = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]

**8. Algorithm to merge an array of size 'n' into another array of size 'm+n'.**

There are two sorted arrays. First one is of size  $m+n$  containing only  $m$  elements. Another one is of size  $n$  and contains  $n$  elements. Merge these two arrays into the first array of size  $m+n$  such that the output is sorted.

Input: array with  $m+n$  elements ( $mPlusN[]$ ).

|   |    |   |    |    |    |    |
|---|----|---|----|----|----|----|
| 2 | NA | 7 | NA | NA | 10 | NA |
|---|----|---|----|----|----|----|

NA => Value is not filled/available in array  $mPlusN[]$ . There should be  $n$  such array blocks.

Input: array with  $n$  elements ( $N[]$ ).

|   |   |    |    |
|---|---|----|----|
| 5 | 8 | 12 | 14 |
|---|---|----|----|

Output:  $N[]$  merged into  $mPlusN[]$  (Modified  $mPlusN[]$ )

|   |   |   |   |    |    |    |
|---|---|---|---|----|----|----|
| 2 | 5 | 7 | 8 | 10 | 12 | 14 |
|---|---|---|---|----|----|----|

## 9. Algorithm to find two repeating numbers in a given array

You are given an array of  $n+2$  elements. All elements of the array are in range 1 to  $n$ . And all elements occur once except two numbers which occur twice. Find the two repeating numbers.

**Example:**

**Input:**

$arr = [4, 2, 4, 5, 2, 3, 1]$

$n = 5$

**Output:**

4 2

**Explanation:**

The above array has  $n + 2 = 7$  elements with all elements occurring once except 2 and 4 which occur twice. So the output should be 4 2.

10. Algorithm to find duplicate elements in O(n) time and O(1) extra space, for a given array of size 'n'
11. Find the index in an array such that the sum of elements at lower indices is equal to the sum of elements at higher indices.
12. Algorithm to find the maximum difference of j - i such that a[j] > a[i], for a given an array of 'n' elements.
13. Algorithm to find the triplet whose sum is X
14. Algorithm to find a sub array whose sum is X
15. Algorithm to find the largest sub array with equal number of 0's and 1's
16. Algorithm to find the number of triangles that can be formed with three different array elements as three sides of triangles, for a given unsorted array of n elements
17. Algorithm to find the smallest integer value that can't be represented as sum of any subset of a given array.
18. Algorithm to find the common element in given three sorted arrays
19. Algorithm to find the contiguous sub-array with maximum sum, for a given array of postive and negative numbers.
20. Given an array of integers, sort the array into a wave like array and return it. (arrange the element into a sequence such that a1>=a2<=a3>=a4<=a5----etc.
21. Algorithm to find the next greater number formed after permuting the digits of given number
22. Algorithm to find the sum of bit difference in all pairs that can be formed from array of n elements.
23. Trapping rain water problem
24. Algorithm to find the minimum number of platforms required for the railway station so that no train waits according to arrival and departure time

25. Rotate 2-Dimensional array
26. Lock and Key problem
27. Rearrange an array so that a[i] becomes a[a[i]] with O(1) extra space
28. Traverse a matrix of integers in spiral form
29. Given an array consisting 0's, 1's and 2's, write a algorithm to sort it
30. Given a positive number X, print all jumping numbers(all adjacent digits in it differ by 1) smaller than or equal to X
31. Given an array and an integer 'k', find the maximum, for each and every contiguous subarray of size 'k'
32. Search an element in a sorted rotated array
33. Find the maximum value of a[j]-a[i]+a[l]-a[k], for every four indices i, j, k, l such that i < j < k < l.

## 2. Linked List

1. Algorithm to find the nth node from end of the linked list
2. Algorithm to find the middle node in a linked list
3. Algorithm to find the intersection point of two linked lists
4. Reversal of linked list
5. Algorithm to detect loop in linked list
6. Algorithm to find starting node of a loop in a linked list
7. Algorithm to check given linked list is palindrome (or) not
8. Algorithm to reverse alternative K nodes in a single linked list
9. Algorithm to clone a linked list with next and random pointer are given...many more

### 3. Stack

1. Reversal of a stack
2. Algorithm to find next greater element on the right side of an array.
3. Implementation of the following operations in stack in O(1) time. Push(), pop(), isEmpty(), isFull() and getMin().
4. Algorithm to find the celebrity in minimum number of questions in a party.
5. Algorithm to the stock span problem is a financial problem where we have a series of 'n' daily price for a stock and we need to calculate span of stock's price for all n days
6. Algorithm to merge overlapping intervals
7. Find the largest rectangular area possible in a given histogram.
8. Given an integer array of size 'n', find the maximum of the minimum's of every window size in the array.
9. Calculate minimum number of bracket reversals needed to make an expression balanced.
10. Design a stack, to find getmin() in O(1) time and O(1) space complexity.
11. Find if an expression has duplicate or not....many more

### 4. Queues

1. Given an array and an integer k, find the maximum for each and every contiguous subarray of size k.
2. Implement LRU Cache.
3. Find the first circular tour that visits all petrol pumps
4. Find the largest multiple of 3.

### 5. Trees

1. Implement in order traversal without stack and recursion
2. Convert a binary tree into its mirror tree

3. Check if a given binary tree is sum tree or not
4. Determine if the given two trees are identical or not
5. Print out all of its root to leaf paths in a given binary tree
6. Find a lowest common ancestor of a given two nodes in a binary search tree
7. Find a lowest common ancestor of a given two nodes in a binary tree
8. Level order traversal in spiral form
9. Convert an arbitrary binary tree to a tree that holds children sum property
10. Find the Diameter of a BST
11. Construct tree from given inorder and post order traversal
12. Convert a Binary Tree to a circular DLL
13. Evaluation of expression tree
14. Print extreme node of each level of Binary Tree in alternative order
15. Print cousins of a given node in Binary Tree
16. Diagonal traversal of Binary Tree
17. Construct tree from ancestor matrix
18. Given a Binary Tree, find vertical sum of the nodes that are in same vertical line.
19. Find multiplication of sums of data of leaves at same level.
20. Given a binary tree, find maximum value we can get by subtracting value of node B from value of node A
21. Print nodes in a top view of Binary Tree.
22. Given a Binary Tree and a number k, remove all nodes that lie only on root to leaf path(s) of length smaller than k.
23. Serialize and deserialize an N-ary tree.
24. Reverse alternate levels of a perfect Binary Tree.
25. Print all nodes that are at distance k from a leaf node.

26. Custom tree problem.
27. Construct complete binary tree from its linked list representation.
28. Find next right nodes of given leafs in a binary tree.
29. Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root.
30. Convert a given tree to its sum tree.
31. Given a binary tree, find out if the tree can be folded or not.
32. Find largest sub tree having identical left and right sub tree.
33. Convert a normal binary search tree to balanced BST.
34. Check if removing an edge can divide a binary tree in the form of n-ary tree.
35. locking and unlocking of resource arranged on the form of n-ary tree.

## 6. Heaps

1. Find K largest (or smallest) elements in array.
2. Tournament tree method using binary heap .
3. Find a Median in a stream of integers.
4. Sort a nearly sorted array(or k sorted).
5. Given array representation of min Heap, convert it to max Heap.
6. Check if a given binary tree is Heap.
7. Find kth largest element in a stream.
8. Print all elements in sorted order from row and column wise sorted matrix.
9. Given n ropes of different length, connect with minimum cost.
10. Given k sorted arrays of size n each, merge them.
11. Design an efficient data structure for given operations find min(), findmax(), deletemin(), Insert(), delete().

## 7. Strings

1. Given a directed graph  $G=(V,E)$ , find whether  $G$  has a cycle.
2. Given an undirected graph  $G=(V,E)$ , find whether  $G$  has a cycle or not.
3. Given a directed graph  $G=(V,E)$ , find whether there is path between two vertices  $v_i$  and  $v_j$ .
4. Given a 2D boolean matrix, find the number of islands.
5. Given a connected undirected graph, find all the articulation points
6. Given an undirected graph, find all the bridges in the graph.
7. Given a directed graph, find all the strongly connected components.
8. Given a weighted directed acyclic graph, find the shortest path from a vertex to all the other vertices.
9. Given a weighted directed acyclic graph, find the longest path from a vertex to all the other vertices.
10. Check whether a given graph is bipartite or not.
11. Find number of connections a person till nth level.

## 8. Bit Manipulation



# Algorithm

## Data Structures

| Data Structure     | Time Complexity |           |           |           |          |           |           |           | Space Complexity |  |
|--------------------|-----------------|-----------|-----------|-----------|----------|-----------|-----------|-----------|------------------|--|
|                    | Average         |           |           |           | Worst    |           |           |           |                  |  |
|                    | Indexing        | Search    | Insertion | Deletion  | Indexing | Search    | Insertion | Deletion  |                  |  |
| Basic Array        | O(1)            | O(n)      | -         | -         | O(1)     | O(n)      | -         | -         | O(n)             |  |
| Dynamic Array      | O(1)            | O(n)      | O(n)      | -         | O(1)     | O(n)      | O(n)      | -         | O(n)             |  |
| Singly-Linked List | O(n)            | O(n)      | O(1)      | O(1)      | O(n)     | O(n)      | O(1)      | O(1)      | O(n)             |  |
| Doubly-Linked List | O(n)            | O(n)      | O(1)      | O(1)      | O(n)     | O(n)      | O(1)      | O(1)      | O(n)             |  |
| Skip List          | O(n)            | O(log(n)) | O(log(n)) | O(log(n)) | O(n)     | O(n)      | O(n)      | O(n)      | O(n log(n))      |  |
| Hash Table         | -               | O(1)      | O(1)      | O(1)      | -        | O(n)      | O(n)      | O(n)      | O(n)             |  |
| Binary Search Tree | -               | O(log(n)) | O(log(n)) | O(log(n)) | -        | O(n)      | O(n)      | O(n)      | O(n)             |  |
| B-Tree             | -               | O(log(n)) | O(log(n)) | O(log(n)) | -        | O(log(n)) | O(log(n)) | O(log(n)) | O(n)             |  |
| Red-Black Tree     | -               | O(log(n)) | O(log(n)) | O(log(n)) | -        | O(log(n)) | O(log(n)) | O(log(n)) | O(n)             |  |
| AVL Tree           | -               | O(log(n)) | O(log(n)) | O(log(n)) | -        | O(log(n)) | O(log(n)) | O(log(n)) | O(n)             |  |

## Heaps

| Heaps                  | Time Complexity |           |             |              |           |            |           |  |
|------------------------|-----------------|-----------|-------------|--------------|-----------|------------|-----------|--|
|                        | Heapify         | Find Max  | Extract Max | Increase Key | Insert    | Delete     | Merge     |  |
| Linked List (sorted)   | -               | O(1)      | O(1)        | O(n)         | O(n)      | O(1)       | O(m+n)    |  |
| Linked List (unsorted) | -               | O(n)      | O(n)        | O(1)         | O(1)      | O(1)       | O(1)      |  |
| Binary Heap            | O(log(n))       | O(1)      | O(log(n))   | O(log(n))    | O(log(n)) | O(log(n))  | O(m+n)    |  |
| Binomial Heap          | -               | O(log(n)) | O(log(n))   | O(log(n))    | O(log(n)) | O(log(n))  | O(log(n)) |  |
| Fibonacci Heap         | -               | O(1)      | O(log(n))*  | O(1)*        | O(1)      | O(log(n))* | O(1)      |  |

## Graphs

| Node / Edge Management | Storage      | Add Vertex   | Add Edge     | Remove Vertex | Remove Edge  | Query  |
|------------------------|--------------|--------------|--------------|---------------|--------------|--------|
| Adjacency list         | O( V  +  E ) | O(1)         | O(1)         | O( V  +  E )  | O( E )       | O( V ) |
| Incidence list         | O( V  *  E ) | O(1)         | O(1)         | O( E )        | O( E )       | O( E ) |
| Adjacency matrix       | O( V ^2)     | O( V ^2)     | O(1)         | O( V ^2)      | O(1)         | O(1)   |
| Incidence matrix       | O( V  *  E )  | O( V  *  E ) | O( E ) |

## Searching

| Algorithm                                                               | Data Structure                            | Time Complexity           |                           |          | Space Comple |
|-------------------------------------------------------------------------|-------------------------------------------|---------------------------|---------------------------|----------|--------------|
|                                                                         |                                           | Average                   | Worst                     | Worst    |              |
| Depth First Search (DFS)                                                | Graph of $ V $ vertices and $ E $ edges   | $O( E  +  V )$            | $O( E  +  V )$            | $O( V )$ |              |
| Breadth First Search (BFS)                                              | Graph of $ V $ vertices and $ E $ edges   | $O( E  +  V )$            | $O( E  +  V )$            | $O( V )$ |              |
| Binary search                                                           | Sorted array of $n$ elements              | $O(\log(n))$              | $O(\log(n))$              | $O(1)$   |              |
| Linear (Brute Force)                                                    | Array                                     | $O(n)$                    | $O(n)$                    | $O(1)$   |              |
| Shortest path by Dijkstra,<br>using a Min-heap as priority queue        | Graph with $ V $ vertices and $ E $ edges | $O(( V  +  E ) \log  V )$ | $O(( V  +  E ) \log  V )$ | $O( V )$ |              |
| Shortest path by Dijkstra,<br>using an unsorted array as priority queue | Graph with $ V $ vertices and $ E $ edges | $O( V ^2)$                | $O( V ^2)$                | $O( V )$ |              |
| Shortest path by Bellman-Ford                                           | Graph with $ V $ vertices and $ E $ edges | $O( V  E )$               | $O( V  E )$               | $O( V )$ |              |

## Sorting

| Algorithm      | Data Structure | Time Complexity |                |                | Worst Case Auxiliary Space Complexity |
|----------------|----------------|-----------------|----------------|----------------|---------------------------------------|
|                |                | Best            | Average        | Worst          |                                       |
| Quicksort      | Array          | $O(n \log(n))$  | $O(n \log(n))$ | $O(n^2)$       | $O(\log(n))$                          |
| Mergesort      | Array          | $O(n \log(n))$  | $O(n \log(n))$ | $O(n \log(n))$ | $O(n)$                                |
| Heapsort       | Array          | $O(n \log(n))$  | $O(n \log(n))$ | $O(n \log(n))$ | $O(1)$                                |
| Bubble Sort    | Array          | $O(n)$          | $O(n^2)$       | $O(n^2)$       | $O(1)$                                |
| Insertion Sort | Array          | $O(n)$          | $O(n^2)$       | $O(n^2)$       | $O(1)$                                |
| Select Sort    | Array          | $O(n^2)$        | $O(n^2)$       | $O(n^2)$       | $O(1)$                                |
| Bucket Sort    | Array          | $O(n+k)$        | $O(n+k)$       | $O(n^2)$       | $O(nk)$                               |
| Radix Sort     | Array          | $O(nk)$         | $O(nk)$        | $O(nk)$        | $O(n+k)$                              |

## 1. Sorting Algorithms

### 1. Bubble Sort

```

static int[] sort(int[] a) {

 for(int i=0;i<a.length-1;i++)

 for(int j=i+1;j<a.length;j++) {

 if(a[i]>a[j]) {

 int s = a[i];
 a[i] = a[j];
 a[j] = s;

 }
 }

}

```

```

 }
 return a;
}

```

## 2. Selection Sort

```

public static void sort(int[] a) {
 for(int i = 0;i<a.length;i++) {
 int min = i;
 for(int j=i+1;j<a.length;j++) {
 if(a[min]>a[j])min= j;
 }
 int swap = a[i];
 a[i] = a[min];
 a[min] = swap;
 }
}

```

## 3. Insertion Sort

```

public static void sort(int[] a) {
 for(int i = 1;i<a.length;i++) {
 int j = i;
 int temp = a[j];
 while(j>0 && a[j-1]>temp) {
 a[j] = a[j-1];
 j--;
 }
 a[j] = temp;
 }
}

```

#### 4. Merge Sort

```
private static void sort(int[] a,int l,int r) {
 if(l < r) {
 int m = (l+r)/2;
 sort(a,l,m);
 sort(a,m+1,r);
 merge(a,l,m,r);
 }
}
```

```
private static void merge(int[] arr, int l, int m, int r) {
```

```
 int n1 = m-l+1;
 int n2 = r-m;

 int[] L = new int[n1];
 int[] R = new int[n2];
 /*Copy data to temp arrays*/
```

```
for (int i=0; i < n1; ++i)
```

```
 L[i] = arr[l + i];
```

```
for (int j=0; j < n2; ++j)
```

```
 R[j] = arr[m + 1+j];
```

```
/* Merge the temp arrays */
```

```
// Initial indexes of first and second subarrays
```

```
int i = 0, j = 0;
```

```
// Initial index of merged subarry array
```

```
int k = l;
```

```
while (i < n1 && j < n2)
```

```
{
```

```
 if (L[i] <= R[j])
```

```
{
```

```
 arr[k] = L[i];
```

```
 i++;
```

```
}
```

```
else
```

```
{
```

```
 arr[k] = R[j];
```

```
 j++;
```

```
}
```

```
 k++;
```

```
}
```

```
/* Copy remaining elements of L[] if any */
```

```
while (i < n1)
```

```
{
```

```
 arr[k] = L[i];
```

```
 i++;
```

```
 k++;
```

```
}
```

```
/* Copy remaining elements of R[] if any */
```

```
while (j < n2)
```

```
{
```

```
 arr[k] = R[j];
```

```
 j++;
```

```
 k++;
```

```
}
```

```
}
```

## 5. Quick sort

```
private static void sort(int[] a, int low, int high) {
```

```
 if(low<high) {
```

```
 int pivot = partition(a,low,high);
```

```
 sort(a,low,pivot-1);
```

```
 sort(a,pivot+1,high);
```

```
}
```

```
}
```

```
private static int partition(int[] a, int low, int high) {
```

```
 int i = low-1;
```

```
 int pivot = high;
```

```
 for(int j=low;j<high;j++) {
```

```
 if(a[j]<=a[pivot]) {
```

```
 i++;
```

```
 int s1= a[j];
```

```
 a[j] = a[i];
```

```
 a[i] = s1;
```

```
}
```

```
 }

 int s2 = a[i+1];
 a[i+1] = a[high];
 a[high] = s2;
 return i+1;
}

6. Heap Sort
public static void sort(int[] a) {

 for(int i = 0;i<a.length;i++) {

 int min = i;

 for(int j=i+1;j<a.length;j++) {
 if(a[min]>a[j])min= j;

 }

 int swap = a[i];
 a[i] = a[min];
 a[min] = swap;

 }
}
```

## Divide & Conquer

7. Find the median of two sorted arrays
8. Count inversions in an array
9. Find majority Element in a sorted array
10. Find the maximum and minimum of an array using minimum number of comparisons
11. The skyline problem
12. Given two binary strings that represent value of two integers, find the product of two strings.
13. Given an array of integers. Find a peak element in it.
14. Find the missing number in Arithmetic Progression
15. Given an array of n points in the plane, find out the closest pair of points in the array.

## Back Tracking

16. Print all permutations of a given string.
17. Find subset of elements that are selected from a given set whose sum adds up to a given number K.
18. Given a set of n integers, divide the set in two subsets of n/2 sizes each such that the difference of the sum of two subsets is as minimum as possible.
19. Solve Sudoku using backtracking.
20. Given a maze, NxN matrix. A rat has to find a path from source to destination. Left top corner is the source and right bottom corner is destination. There are few cells which are blocked, means rat can't enter into those cells.

## Pattern searching

21. Given a text and a pattern, find all occurrences of pattern in a given text. Using naive approach.
22. Given a text and a pattern, find all occurrences of pattern in a given text. Using Rabin-Karp algorithm.
23. Given a text and a pattern, find all occurrences of pattern in a given text. Using Finite automata approach.
24. Given a text and a pattern, find all occurrences of pattern in a given text. Using Boyer moore algorithm.
25. Given a text and a pattern, find all occurrences of pattern in a given text. Using KMP algorithm.
26. Given a string, find the longest sub string which is palindrome using manacher's algorithm
27. Find all occurrences of a given word in a matrix.

## Greedy Algorithms

28. Given an array of jobs with different time intervals. Find the minimum time to finish all jobs.
29. Given a universe of n elements, collection of subsets. Find a minimum cost sub collection that covers all elements.
30. Given n cities and distances between every pair of cities, select k cities to place warehouses, such that the maximum distance of a city to a warehouse is minimized.

## Dynamic Programming

31. Find the length of the longest sub sequence of a given sequence such that all elements of the sub sequence are sorted in increasing order.
32. Given two sequences, find the length of longest sub sequence present in both of them.
33. Given a cost matrix and a position (m, n) , Find cost of minimum cost path to reach (m, n) from (0, 0).
34. Coin change problem.
35. Find the length of the longest palindrome sub sequence.
36. Find th sum of maximum sum sub sequence of the given array.
37. You have a rectangular grid of dimension 2 x n. You need to find out the maximum sum such that no two chosen numbers are adjacent , vertically, diagonally (or) horizontally.
38. Given an array A with n elements and array B with m elements. With m you have to insert (n-m) zero's in between array B such that the dot product of array A and array B is maximum.
39. Transform a string into palindrome on removing at most k characters from it.
40. Find the longest even length sub string such that sum of first and second half is same..
41. Count number of ways to reach a given score in a game.
42. Compute sum of digits in all number from 1 to n.
43. Collect maximum points in a grid using two traversals
44. Given a 2xn board and titles of size 2x1, count the number of ways to tile he given board using the 2x1 tiles..
45. Count the number of ways we can parenthesize the expression so that the value of expression evaluates to true.
46. Given a Binary Tree, find size of the Largest Independent Set(LIS) in it.

47. There are n stairs, a person standing at the bottom wants to reach the top. The person can climb either 1 stair or 2 stairs at a time.  
Count the number of ways, the person can reach the top.
48. Find total number of non-decreasing numbers with n digits.
49. Egg dropping problem.
50. Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces.
51. Given N jobs where every job is represented by Start Time, Finish Time, Profit or Value Associated. Find the maximum profit subset of jobs such that no two jobs in the subset overlap.
52. Box stacking problem.
53. Given an input string and a dictionary of words, find out if the input string can be segmented into a space-separated sequence of dictionary words.
54. Given a binary matrix, find out the maximum size square sub-matrix with all 1s.
55. Find the maximum coins you can collect by bursting the balloons wisely.

## Databases

### 1. Interview questions

able - EmployeeDetails

| EmpId | FullName | ManagerId | DateOfJoining |
|-------|----------|-----------|---------------|
|       |          |           |               |

|     |              |     |            |
|-----|--------------|-----|------------|
| 121 | John Snow    | 321 | 01/31/2014 |
| 321 | Walter White | 986 | 01/30/2015 |
| 421 | Kuldeep Rana | 876 | 27/11/2016 |

**Table - EmployeeSalary**

| EmpId | Project | Salary |
|-------|---------|--------|
| 121   | P1      | 8000   |
| 321   | P2      | 1000   |
| 421   | P1      | 12000  |

1. Write a SQL query to fetch the count of employees working in project 'P1'.

Ans. Here, we would be using aggregate function count() with the SQL where clause-

```
SELECT COUNT(*) FROM EmployeeSalary WHERE Project = 'P1';
```

2. Write a SQL query to fetch employee names having salary greater than or equal to 5000 and less than or equal 10000.

Ans. Here, we will use BETWEEN in the 'where' clause to return the empId of the employees with salary satisfying the required criteria and then use it as subquery to find the fullName of the employee from EmployeeDetails table.

```
SELECT FullName
FROM EmployeeDetails
WHERE EmpId IN
(SELECT EmpId FROM EmployeeSalary
WHERE Salary BETWEEN 5000 AND 10000);
```

3. Write a SQL query to fetch project-wise count of employees sorted by project's count in descending order.

Ans. The query has two requirements - first to fetch the project-wise count and then to sort the result by that count. For project wise count, we will be using GROUPBY clause and for sorting, we will use ORDER BY clause on the alias of the project-count.

```
SELECT Project, count(EmpId) EmpProjectCount
FROM EmployeeSalary
GROUP BY Project
ORDER BY EmpProjectCount DESC;
```

4. Write a query to fetch only the first name(string before space) from the FullName column of EmployeeDetails table.

Ans. In this question, we are required to first fetch the location of the space character in the FullName field and then extract the first name out of the FullName field. For finding the location we will use LOCATE method in MySQL and CHARINDEX in SQL SERVER and for fetching the string before space, we will use SUBSTRING OR MID method.

**MySQL- Using MID**

```
SELECT MID(FullName, 0, LOCATE(' ', FullName)) FROM EmployeeDetails;
```

### SQL Server-Using SUBSTRING

```
SELECT SUBSTRING(FullName, 0, CHARINDEX(' ', FullName)) FROM EmployeeDetails;
```

Also, we can use LEFT which returns the left part of a string till specified number of characters.

```
SELECT LEFT(FullName, CHARINDEX(' ', FullName) - 1) FROM EmployeeDetails;
```

5. Write a query to fetch employee names and salary records.  
Return employee details even if the salary record is not present for the employee.

Ans. Here, we can use left join with EmployeeDetail table on the left side.

```
SELECT E.FullName, S.Salary
FROM EmployeeDetails E LEFT JOIN EmployeeSalary S
ON E.EmpId = S.EmpId;
```

6. Write a SQL query to fetch all the Employees who are also managers from EmployeeDetails table.

Ans. Here, we have to use Self-Join as the requirement wants us to analyze the EmployeeDetails table as two different tables, each for Employee and manager records.

```
SELECT DISTINCT E.FullName
FROM EmpDetails E
INNER JOIN EmpDetails M
ON E.EmpID = M.ManagerID;
```

7. Write a SQL query to fetch all employee records from EmployeeDetails table who have a salary record in EmployeeSalary table.

Ans. Using 'Exists'-

```
SELECT * FROM EmployeeDetails E
WHERE EXISTS
(SELECT * FROM EmployeeSalary S WHERE E.EmpId = S.EmpId);
```

8. Write a SQL query to fetch duplicate records from a table.

Ans. In order to find duplicate records from table we can use GROUP BY on all the fields and then use HAVING clause to return only those fields whose count is greater than 1 i.e. the rows having duplicate records.

```
SELECT EmpId, Project, Salary, COUNT(*)
FROM EmployeeSalary
GROUP BY EmpId, Project, Salary
HAVING COUNT(*) > 1;
```

9. Write a SQL query to remove duplicates from a table without using temporary table.

Ans. Using Group By and Having clause-

```
DELETE FROM EmployeeSalary
WHERE EmpId IN (
SELECT EmpId
FROM EmployeeSalary
GROUP BY Project, Salary
HAVING COUNT(*) > 1));
```

Using rowId in Oracle-

```
DELETE FROM EmployeeSalary
WHERE rowid NOT IN
(SELECT MAX(rowid) FROM EmployeeSalary GROUP BY EmpId);
```

## 10. Write a SQL query to fetch only odd rows from table.

Ans. This can be achieved by using Row\_number in SQL server-

```
SELECT E.EmpId, E.Project, E.Salary
FROM (
 SELECT *, Row_Number() OVER(ORDER BY EmpId) AS RowNumber
 FROM EmployeeSalary
) E
WHERE E.RowNumber % 2 = 1
```

## 11. Write a SQL query to fetch only even rows from table.

Ans. Using the same Row\_Number() and checking that the remainder when divided by 2 is 0-

```
SELECT E.EmpId, E.Project, E.Salary
FROM (
 SELECT *, Row_Number() OVER(ORDER BY EmpId) AS RowNumber
 FROM EmployeeSalary
) E
WHERE E.RowNumber % 2 = 0
```

## 12. Write a SQL query to create a new table with data and structure copied from another table.

Ans. Using SELECT INTO command-

```
SELECT * INTO newTable FROM EmployeeDetails;
```

13. Write a SQL query to create an empty table with same structure as some other table.

Ans. Using SELECT INTO command with False 'WHERE' condition-

```
SELECT * INTO newTable FROM EmployeeDetails WHERE 1 = 0;
```

This can also be done using MySQL 'Like' command with CREATE statement-

```
CREATE TABLE newTable LIKE EmployeeDetails;
```

14. Write a SQL query to fetch common records between two tables.

Ans. Using INTERSECT-

```
SELECT * FROM EmployeeSalary
INTERSECT
SELECT * FROM ManagerSalary
```

15. Write a SQL query to fetch records that are present in one table but not in another table.

Ans. Using MINUS-

```
SELECT * FROM EmployeeSalary
MINUS
SELECT * FROM ManagerSalary
```

**16. Write a SQL query to find current date-time.**

Ans. MySQL-

```
SELECT NOW();
```

SQL Server-

```
SELECT getdate();
```

Oracle-

```
SELECT SYSDATE FROM DUAL;
```

**17. Write a SQL query to fetch all the Employees details from EmployeeDetails table who joined in Year 2016.**

Ans. Using BETWEEN for the date range '01-01-2016' AND '31-12-2016'-

```
SELECT * FROM EmployeeDetails
WHERE DateOfJoining BETWEEN '01-01-2016' AND date '31-12-2016';
```

Also, we can extract year part from the joining date (using YEAR in MySQL)-

```
SELECT * FROM EmployeeDetails
WHERE YEAR(DateOfJoining) = '2016';
```

## 18. Write a SQL query to fetch top n records?

Ans. In mySQL using LIMIT-

```
SELECT * FROM EmployeeSalary ORDER BY Salary DESC LIMIT N
```

In SQL server using TOP command-

```
SELECT TOP N * FROM EmployeeSalary ORDER BY Salary DESC
```

In Oracle using ROWNUM-

```
SELECT * FROM (SELECT * FROM EmployeeSalary ORDER BY Salary DESC)
WHERE ROWNUM <= 3;
```

## 19. Write SQL query to find the nth highest salary from table.

Ans. Using Top keyword (SQL Server)-

```
SELECT TOP 1 Salary
FROM (
 SELECT DISTINCT TOP N Salary
 FROM Employee
 ORDER BY Salary DESC
)
ORDER BY Salary ASC
```

Using limit clause(mySQL)-

```
SELECT Salary FROM Employee ORDER BY Salary DESC LIMIT N-1,1;
```

20. Write SQL query to find the 3rd highest salary from table without using TOP/limit keyword.

Ans. The below SQL query make use of correlated subquery wherein in order to find the 3rd highest salary the inner query will return the count of till we find that there are two rows that salary greater than other distinct salaries.

```
SELECT Salary
FROM EmployeeSalary Emp1
WHERE 2 = (
 SELECT COUNT(DISTINCT (Emp2.Salary))
 FROM EmployeeSalary Emp2
 WHERE Emp2.Salary > Emp1.Salary
)
```

For nth highest salary-

```
SELECT Salary
FROM EmployeeSalary Emp1
WHERE N-1 = (
 SELECT COUNT(DISTINCT (Emp2.Salary))
 FROM EmployeeSalary Emp2
 WHERE Emp2.Salary > Emp1.Salary
)
```

- 2. SQL Queries Basics**
- 3. SELECT statement and syntax**
- 4. SQL Data Types**
- 5. WHERE operators: comparison operators, LIKE, BETWEEN, IN**
- 6. Scalar Functions**
- 7. Aggregate Functions (SUM, COUNT, MIN, MAX, AVG)**
- 8. GROUP BY**
- 9. ORDER BY**
- 10. JOINS (INNER JOIN, RIGHT and LEFT JOIN, OUTER JOIN, CROSS JOIN)**
- 11. Set Theory (INTERSECT, UNION, MINUS)**
- 12. Subqueries**
- 13. INSERT, UPDATE, DELETE**
- 14. DML vs DDL vs DCL**
- 15. DDL: CREATE, DROP, ALTER**
- 16. Constraints**
- 17. Indexes**
- 18. DCL privileges: GRANT, REVOKE**
- 19. Transactions**
- 20. Views**
- 21. Triggers**
- 22. Cursors**

