1. Define Java Architecture:

Java combines both the approaches:

* Compilation
* Interpretation

Source Code 🡪 Java Compiler 🡪 Byte Code 🡪 JVM1🡪Windows OS, JVM2🡪Unix OS

JRE: JVM + Class Lib + Other Supporting components

1. JVM tasks (converts byte code to machine code)

Class Loader 🡪 Byte Code Verifier 🡪 JVM , JIT

1. Class Loader:

* Boot Strap Class Loader: loads classes from bootstrap classpath(rt.jar)
* Extension Classloader: loads class ext folder(jre\lib\ext)
* Application Classloader: loading of application level, mentioned in classpath and env var etc..

1. Byte Code Verifier: No unauthorized access to memory, no stack overflow, no illegal data conversion
2. Java Memory Model:

* Heap Area : Created during startup. Represents Runtime data area. Memory allocated for all the class instances and arrays. Can be fixed or dynamic. Throws “Out of memory error”
* Method Area and runtime constant pool: Stores per class structure. Logically part of heap but cannot be garbage collected. Throws “Out of memory error”
* JVM Stack: Private stack for each thread. Stores local var. Throes “Stack OverFlow exception”. Can be fixed or dynamic. Can also throw “Out of memory error” when dynamic increase of stack and have insufficient memory.
* Native method stack: Collection of C, C++ lib which r needed for execution.

1. Diff b/w JDK,JRE,JVM

JDK ( JRE ( JVM+Lib) + Dev Tools)

1. Static Import:

Priority :

1. Local static method
2. import static package1.A.test; //Single Static import //
3. import static package1.B.\*; //Static import on demand

error:

import static package1.A.test; //Single Static import //

import static package1.B.test; //Single Static import //compile time error

1. 2 Interface having same method and Implementation class will implement without error.
2. Polymorphism
3. Constructor.java
4. Serialization :
5. Any final field reference is serialized even if transient mentioned.
6. Throws : java.io.NotSerializableException.
7. HashMap uses transient.
8. Cannot serialize Logger bx it does not implement Serialization.
9. Static, transient field does not get serialize.
10. How we can avoid serialization? Override method (writeObject, readObject and throw NotSerializableException.
11. Exception Handling
12. Basics.
13. Can “Out of Memory be catched”
14. Sample program
15. ARM : AutoClosableResourcManagement
16. Collection:
17. HashMap implementation :

* Array of nodes (int hash, k key, v value, Node<k,v> next
* Initial capacity:16
* Load factor=0.75, should alwz b b/w 0 and 1.
* Time complexity 🡪 capacity (no. of buckets)+size
* Rehashing is a process of increasing the capacity multipled by 2.

1. syncronizedMap

* Collections.syncronizedMap(new HashMap());
* Iterators of this are fail-fast. Throws “ConcurrentModificationException”
* In Hashtable, the hashCode method only gets invoked when you use the Object as the key to a Hashtable. It is not used when the Object is a Hashtable value. Most of the time your Hashtable keys are simple Strings, so you rarely need to write custom equals and hashCode methods. But in the case of HashSet, to help you check for duplicate Objects, we need a custom equals and hashCode method because here our Objects act as both key and value.
* *Lets see how HashMap uses the hash code.*

In HashMap, when you try to get the object using get(key) method, the HashMap first gets the hash code of the key (remember that key is an object in the HashMap and it should have hashCode() method overridden) and generates again the hashcode to just defend against poor quality hash functions. Using this hashcode, an index for hash code will be created and this index is used to look up the entry table which is where all the hash entries are available. Now, once you get the Entry object from the Entry table using index, check if the key of the Entry object is same as the key we sent as parameter to get method. If yes, then return the value of the Entry object. If no, then go to next Entry object and continue the same.

Below is the code in the get method of HashMap

int hash = hash(key.hashCode());

for (Entry e = table[indexFor(hash, table.length)];

e != null;

e = e.next) {

Object k;

if (e.hash == hash && ((k = e.key) == key || key.equals(k)))

return e.value;

}

Given:

11. public class Person {

12. private String name, comment;

13. private int age;

14. public Person(String n, int a, String c) {

15. name = n; age = a; comment = c;

16. }

17. public boolean equals(Object o) {

18. if (! (o instanceof Person)) return false;

19, Person p = (Person)o;

20. return age == p.age && name.equals(p.name);

21. }

22. }

What is the appropriate definition of the hashCode method in class Person?

A. return super.hashCode();

B. return name.hashCode() + age \* 7;

C. return name.hashCode() + comment.hashCode() / 2;

D. return name.hashCode() + comment.hashCode() / 2 - age \* 3;

Can you guess? Yes, you are right. Its B. Why is it B? Before looking at hashCode method, look at equals method. In equals method the comparision is made between person's age and name to check if the two objects are equal. So we need to consider only person's age and name when generating hashCode.

Got it? Now again go back to previous example of Student class. In the "equals" method or "hashCode" method I didn't use "marks" field to compare or generate the hash code. This I have indicated in the note above that "while implementing the hashcode() method the fields that are present in the hashcode() should not be the one which could change the state of object.". Since "marks" field can change the state of the object, I didn't use marks field in generating hash code. See below example.

1. Threading:
2. Basics
3. Can run() be called outside
4. Thread join
5. Inter thread Communication (classic +BlockinQueue)
6. DeadLock
7. How we can analyze deadlock: taking thread dump. Kill -3, ctrl +Break
8. Locking + Syncronization
   * Volatile
   * Atomic var
   * Explicit locking via (java.util.concurrent.lock.Lock Interface, classes : ReentrantLock, ReentrantReadWriteLock
9. Static/no-static syncrinize methods/syncrnizedblocks
10. Rentrant Nature: holds locks when SM method calls another SM emthods from inside.
11. NullPointerException : if object used in SM method is null.
12. Race Condition :

* Check and act (Contains key then put, need to wrap in s block to avoid race cond.
* Read modify write (++)

1. Livelock
2. Starvation.
3. RentrantLock : locking mechanism
4. Concurrent Collections :
   1. Concurrent HashMap
   2. CopyOnWriteArrayList/ CopyOnWriteArraySet
   3. BlockingQueue : ArrayBlockingQueue/LinkedBlockingQueue
   4. DeQueue &BlockingDeque
5. How we can done Class level locking : by adding synchronized in static methos.
6. What happens when notify is called and no thread is waiting?
7. Alwaz put wait method in loop.
8. How to handle Rejected task

**Bubble Sort Performance and Complexity**

1. Bubble sort belongs to O(n2) sorting algorithms, which makes it quite inefficient for sorting large data volumes.
2. Bubble sort is both [**stable**](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability) and [**adaptive**](https://en.wikipedia.org/wiki/Adaptive_sort).
3. In the case of nearly sorted data, bubble sort takes O(n) time, but requires at least 2 passes through the data.
4. It can be practical if the input is usually in sort order but may occasionally have some out-of-order elements nearly in position.
5. Bubble sort should be avoided in the case of large collections.
6. It will not be efficient in the case of a reverse-ordered collection.

**Quicksort algorithm** is one of the most used sorting algorithm, especially to sort large lists/arrays. Quicksort is a **divide and conquer algorithm**, which means original array is divided into two arrays, each of them is sorted individually and then sorted output is merged to produce the sorted array. On the average, **it has O(n log n) complexity**, making quicksort suitable for sorting big data volumes.

**Quicksort algorithm**

The basic idea of Quicksort algorithm can be described as these steps:

If the array contains only one element or zero elements then the array is sorted. If the array contains more then one element then:

Steps to implement Quick sort algorithm in place:  
  
1) Choose an element, called pivot, from the list or array. Generally pivot is the middle element of array.  
  
2) Reorder the list so that all elements with values less than the pivot come before the pivot, and all elements with values greater than the pivot come after it (equal values can go either way). This is also known as *partitioning*. After partitioning the pivot is in its final position.  
  
3) Recursively apply the above steps to the sub-list of elements with smaller values and separately the sub-list of elements with greater values. If the array contains only one element or zero elements then the array is sorted.

### Import points about Quicksort algorithm

Now we know how quick sort works and how to implement quicksort in Java, its time to revise some of the important points about this popular sorting algorithm.  
  
1) QuickSort is a divide and conquer algorithm. Large list is divided into two and sorted separately (conquered), sorted list is merge later.  
  
2) On "in-place" implementation of quick sort, list is sorted using same array, no additional array is required. Numbers are re-arranged pivot, also known as partitioning.  
  
3) Partitioning happen around pivot, which is usually middle element of array.  
  
4) Average case time complexity of Quicksort is O(n log n) and worst case time complexity is O(n ^2), which makes it one of the fasted sorting algorithm. Interesting thing is it's worst case performance is equal to [Bubble Sort](http://javarevisited.blogspot.sg/2014/08/bubble-sort-algorithm-in-java-with.html) :)  
  
5) Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion.  
  
6) Quicksort is also a good example of algorithm which makes best use of CPU caches, because of it's divide and conquer nature.  
  
7) In Java, Arrays.sort() method uses quick sort algorithm to sort array of primitives. It's different than our algorithm, and uses two pivots. Good thing is that it perform much better than most of the quicksort algorithm available on internet for different data sets, where traditional quick sort perform poorly. One more reason, not to reinvent the wheel but to use the library method, when it comes to write production code.  
  
  
That's all about **Quicksort sorting algorithm in Java**. It is one of the must know algorithm for all level of Java programmers, not that you need it often to implement it but to do well on interviews and use the lesson learned while implementing quick sort in Java. In our example, we have implemented quicksort "in-place", which is what you should do if asked to write quicksort in Java. Remember as Java programmer, you don't need to write your own implementation as library implementation are much better implemented and tested. You should use  Arrays.sort()  method to sort your array instead of writing your own sort method. One more reason of using library method is that they are usually improved over different version, and can take advantage of new machine instructions or native improvement.