1.What is race condition

<http://tutorials.jenkov.com/java-concurrency/race-conditions-and-critical-sections.html>

2.Diff between encryption vs encoding vs hashing ?

<https://auth0.com/blog/how-secure-are-encryption-hashing-encoding-and-obfuscation/#What-is-Encoding->

3.How to recover from deadlock?

<http://www.hexainclude.com/recovery-from-deadlock/>

4. Top K repeated elements

public List<Integer> topKFrequent(int[] nums, int k) {

List<Integer>[] bucket = new List[nums.length + 1];

Map<Integer, Integer> frequencyMap = new HashMap<Integer, Integer>();

for (int n : nums) {

frequencyMap.put(n, frequencyMap.getOrDefault(n, 0) + 1);

}

for (int key : frequencyMap.keySet()) {

int frequency = frequencyMap.get(key);

if (bucket[frequency] == null) {

bucket[frequency] = new ArrayList<>();

}

bucket[frequency].add(key);

}

List<Integer> res = new ArrayList<>();

for (int pos = bucket.length - 1; pos >= 0 && res.size() < k; pos--) {

if (bucket[pos] != null) {

res.addAll(bucket[pos]);

}

}

return res;

}

5.Convert array to List

int[] arr = { 1, 2, 3, 4, 5 };

List<Integer> list = Arrays.stream(arr) // IntStream

.boxed() // Stream<Integer>

.collect(Collectors.toList());

6.Convert Integer to binary representation

public class Convert

{

public static void main(String[] args)

{

int n, count = 0, a;

String x = "";

Scanner s = new Scanner(System.in);

System.out.print("Enter any decimal number:");

n = s.nextInt();

while(n > 0)

{

a = n % 2;

if(a == 1)

{

count++;

}

x = a + "" + x;

n = n / 2;

}

System.out.println("Binary number:"+x);

System.out.println("No. of 1s:"+count);

}

}

7.Syncrhonized vs lock

<https://github.com/bryanduxbury/sync_vs_lock/blob/master/LockVsSync.java>

<https://medium.com/liveramp-engineering/java-performance-synchronized-vs-lock-301130e62f47>

8. ArrayList vs LinkedList

1. Insertions are easy and fast in LinkedList as compared to ArrayList because there is no  
   risk of resizing array and copying content to new array if array gets full which makes  
   adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList  
   in Java. ArrayList also needs to be update its index if you insert something anywhere except  
   at the end of array.
2. Removal also better in LinkedList than ArrayList due to same reasons as insertion.
3. LinkedList has more memory overhead than ArrayList because in ArrayList each index only  
   holds actual object (data) but in case of LinkedList each node holds both data and address  
   of next and previous node.
4. Both LinkedList and ArrayList require O(n) time to find if an element is present or not. However we can do Binary Search on ArrayList if it is sorted and therefore can search in O(Log n) time

9.Builder Pattern

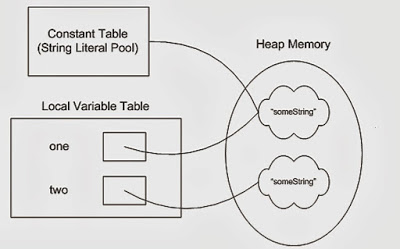
<https://javarevisited.blogspot.com/2012/06/builder-design-pattern-in-java-example.html>

10. **how to write thread safe class or code in Java**

<https://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html>

11. Why is string immutable or final in Java

1) Imagine String pool facility without making string immutable , its not possible at all because in case of string pool one string object/literal e.g. "Test" has referenced by many reference variables, so if any one of them change the value others will be automatically gets affected i.e. lets say  
  
String A = "Test"  
String B = "Test"  
  
Now String B called, "Test".toUpperCase() which change the same object into "TEST", so A will also be "TEST" which is not desirable. Here is a nice diagram which shows how String literals are created in heap memory and String literal pool.

[](https://1.bp.blogspot.com/-4AB58TuuluE/VmLVMNKK4zI/AAAAAAAAENo/Bbkx5wwXlRA/s1600/String+in+Java+is+Immutable.jpg)

2) String has been widely used as parameter for many Java classes e.g. for opening network connection, you can pass hostname and port number as string, you can pass database URL as a string for opening database connection, you can [open any file in Java](http://javarevisited.blogspot.sg/2012/07/read-file-line-by-line-java-example-scanner.html) by passing the name of the file as argument to File I/O classes.  
  
In case, if String is not immutable, this would lead serious security threat, I mean someone can access to any file for which he has authorization, and then can change the file name either deliberately or accidentally and gain access to that file. Because of immutability, you don't need to worry about that kind of threats. This reason also gels with, **Why String is final in Java**, by making java.lang.String final, Java designer ensured that no one overrides any behavior of String class.

12. Concurrent Hashmap

1. ConcurrentHashMap allows concurrent read and thread-safe update operation.

2. During the update operation, ConcurrentHashMap only locks a portion of Map instead of whole Map.

3. The concurrent update is achieved by internally dividing Map into the small portion which is defined by concurrency level.

4. Choose concurrency level carefully as a significantly higher number can be a waste of time and space and the lower number may introduce thread contention in case writers over number concurrency level.

5. All operations of ConcurrentHashMap are [thread-safe](http://javarevisited.blogspot.com/2012/12/how-to-create-thread-safe-singleton-in-java-example.html).

6. Since ConcurrentHashMap implementation doesn't lock whole Map, there is chance of read overlapping with update operations like put() and remove(). In that case result returned by get() method will reflect most recently completed operation from there start.

7. Iterator returned by ConcurrentHashMap is weekly consistent, [fail-safe](http://javarevisited.blogspot.com/2012/02/fail-safe-vs-fail-fast-iterator-in-java.html) and never throw ConcurrentModificationException. In Java.

8. ConcurrentHashMap doesn't allow null as key or value.

9. You can use ConcurrentHashMap in place of [Hashtable](http://javarevisited.blogspot.com/2010/10/difference-between-hashmap-and.html) but with caution as CHM doesn't lock whole Map.

10. During putAll() and clear() operations, the concurrent read may only reflect insertion or deletion of some entries.

13. <https://javarevisited.blogspot.com/2017/08/top-10-java-concurrenthashmap-interview.html>

### 14. What is different between Iterator and ListIterator?

* We can use Iterator to traverse Set and List collections whereas ListIterator can be used with Lists only.
* Iterator can traverse in forward direction only whereas ListIterator can be used to traverse in both the directions.
* ListIterator inherits from Iterator interface and comes with extra functionalities like adding an element, replacing an element, getting index position for previous and next elements.

15. Method reference

<https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>

16. <https://onlyfullstack.blogspot.com/2019/07/java-8-feature-interview-questions.html>

17. <https://github.com/eugenp/tutorials/tree/master/core-java-modules/core-java-streams-2>

18.How Hibernate Implements Cache internally.

<https://docs.jboss.org/hibernate/orm/5.1/userguide/html_single/chapters/caching/Caching.html>

<https://medium.com/@skywalkerhunter/speeding-up-your-hibernate-with-cache-level-2-ehcache-in-1-min-28255622bf6f>

<https://howtodoinjava.com/spring-boot2/spring-boot-cache-example/>

<https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-caching>

<https://docs.spring.io/spring/docs/5.0.0.BUILD-SNAPSHOT/spring-framework-reference/html/cache.html>

<https://dzone.com/articles/spring-hibernate-ehcache-caching>

<https://www.javatpoint.com/hibernate-second-level-cache>

19.Class loaders

<https://www.baeldung.com/java-classloaders>

20. **What is starter dependency in Spring Boot?**

21.Conditional

<https://www.baeldung.com/spring-boot-custom-auto-configuration>

22.Eureka

<https://dzone.com/articles/microservice-spring-cloud-eureka-server-configurat>

<https://www.baeldung.com/spring-cloud-netflix-eureka>

23.Spring scope

<https://howtodoinjava.com/spring-core/spring-bean-scopes/#singleton>

24.Spring Singleton

<https://dzone.com/articles/an-interview-question-on-spring-singletons>

25.Proxy Design Pattern

<https://howtodoinjava.com/design-patterns/structural/proxy-design-pattern/>

baeldung.com/java-proxy-pattern

26.Hiberante Lazy loading

<https://www.baeldung.com/hibernate-lazy-eager-loading>

27.Decorator Pattern

<https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/>

28.Association vs Aggregation vs composition

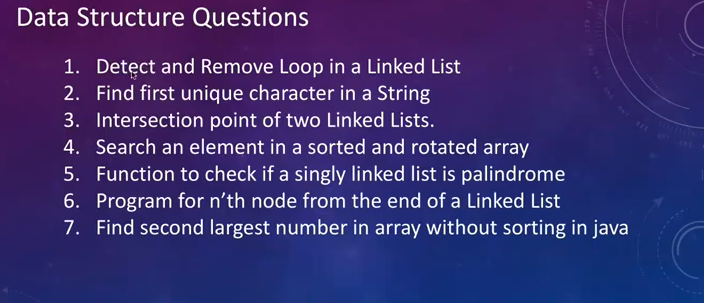
<https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/>

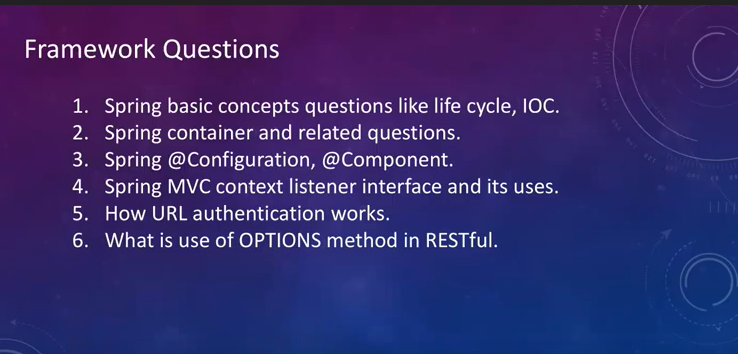
<https://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html>

29.Externalizable interface

30.Intersection of two linked lists

<https://leetcode.com/problems/intersection-of-two-linked-lists/discuss/49785/Java-solution-without-knowing-the-difference-in-len!>





 public static void print2largest(int arr[],

                                     int arr\_size)

    {

        int i, first, second;

        /\* There should be atleast two elements \*/

        if (arr\_size < 2)

        {

            System.out.print(" Invalid Input ");

            return;

        }

        first = second = Integer.MIN\_VALUE;

        for (i = 0; i < arr\_size ; i++)

        {

            /\* If current element is smaller than

            first then update both first and second \*/

            if (arr[i] > first)

            {

                second = first;

                first = arr[i];

            }

            /\* If arr[i] is in between first and

               second then update second  \*/

            else if (arr[i] > second && arr[i] != first)

                second = arr[i];

        }

        if (second == Integer.MIN\_VALUE)

             System.out.print("There is no second largest"+

                                 " element\n");

        else

             System.out.print("The second largest element"+

                                      " is "+ second);

    }

    /\* Driver program to test above function \*/

    public static void main(String[] args)

    {

            int arr[] = {12, 35, 1, 10, 34, 1};

            int n = arr.length;

            print2largest(arr, n);

    }

}

1. **import** java.util.Scanner;
2. **public** **class** SecondLargest\_Smallest
3. {
4. **public** **static** **void** main(String[] args)
5. {
6. **int** n, temp;
7. Scanner s = **new** Scanner(System.in);
8. System.out.print("Enter no. of elements you want in array(Minimum 2):");
9. n = s.nextInt();
10. **int** a[] = **new** **int**[n];
11. System.out.println("Enter all the elements:");
12. **for** (**int** i = 0; i < n; i++)
13. {
14. a[i] = s.nextInt();
15. }
16. **for** (**int** i = 0; i < n; i++)
17. {
18. **for** (**int** j = i + 1; j < n; j++)
19. {
20. **if** (a[i] > a[j])
21. {
22. temp = a[i];
23. a[i] = a[j];
24. a[j] = temp;
25. }
26. }
27. }
28. System.out.println("Second Largest:"+a[n-2]);
29. System.out.println("Smallest:"+a[0]);
30. }
31. }

31.How to enable HTTPS in Spring boot.

<https://www.baeldung.com/spring-boot-https-self-signed-certificate>

<https://www.thomasvitale.com/https-spring-boot-ssl-certificate/>

32.Dependency injection how it works internally

Spring does a lot of things, but dependency injection itself is actually a surprisingly simple mechanism.

It starts with having a registry for classes that are available for injection. Classes that are added to this registry are examined using reflection. A DI framework will look for relevant annotations and constructors to determine how to construct instances of the classes and also what other dependencies these classes may need.

The registry also keeps track of already created instances so they can be re-used. Re-using instances involves scoping, which determines when an instance can be re-used. With singletons (the default for Spring) instances can be re-used without restriction.

To create an instance of class with dependencies, reflection is used to create an instance. If there are any dependencies required, those are created first (if not already created) potentially triggering a lot of recursive creation of instances. If any of the dependencies cannot be created or there are multiple possible candidates, the framework can throw an exception to indicate a problem in your configuration.

A simple example, let's say we have an Injector class that acts as both registry of classes and as a means to create new instances.

We register a few classes:

injector.register(Database.class);

injector.register(EmployeeDao.class);

Let's assume the Database class has no further dependencies, and EmployeeDao has a dependency on Database:

class EmployeeDao {

@Inject Database db;

}

The injector, by means of reflection, knows that EmployeeDao has a dependency on Database. When we ask the injector for an instance of EmployeeDao the following happens:

EmployeeDao employeeDao = injector.getInstance(EmployeeDao.class);

1) A check is done if there already exists an instance of EmployeeDao, if so it is returned.

2) If not, a check is done to see what is needed to construct EmployeeDao, in this case it needs a Database. The injector calls itself recursively with:

Database database = injector.getInstance(Database.class);

2a) Again a check is done if an instance of Database is already available.

2b) There are no further dependencies required in order to construct Database so the injector calls Database.class.newInstance() and keeps track of it.

2c) A Database instance is returned.

3) With the Database instance available, the injector can now construct the EmployeeDao: EmployeeDao.class.newInstance() -- with the help of reflection, the field database is injected with the Database instance.

4) The EmployeeDao instance, now fully injected, is returned.

This is fairly direct way of obtaining an instance of a class, however this is at the core how DI frameworks like Spring work. More advanced features require creation of dynamic proxies and the use of AOP, but DI itself boils down to using reflection to construct instances automatically.

Qualifier

<https://www.baeldung.com/spring-qualifier-annotation>

<http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html>

Aware Interfaces In Spring

<https://springframework.guru/using-spring-aware-interfaces/>

<https://github.com/spring-framework-guru/sfg-blog-posts/tree/master/springawareinterfaces>

Composite Primary key

<https://www.callicoder.com/hibernate-spring-boot-jpa-composite-primary-key-example/>

<https://stackoverflow.com/questions/14037404/java-read-large-text-file-with-70million-line-of-text>

<https://www.javaguides.net/2018/10/free-open-source-projects-using-spring-boot.html>

<https://www.youtube.com/csdojo>

<https://www.youtube.com/backtobackswe>

<https://www.youtube.com/nickwhite>

<https://leetcode.com/explore/learn/card/binary-search/136/template-analysis/935/>

<https://github.com/donnemartin/system-design-primer>

<https://www.youtube.com/watch?v=Dbxzw0cpxBU>

Synchronization

<https://www.baeldung.com/java-synchronized>

Concurrent locks

<https://www.baeldung.com/java-concurrent-locks>

Fail fast vs fail safe iterator

Occurrences of characters in string

Hash Code and Equals

<https://www.baeldung.com/java-equals-hashcode-contracts>

Spring Web contexts

<https://www.baeldung.com/spring-web-contexts>

Reloading Properties

<https://www.baeldung.com/spring-reloading-properties>