Java interview

Sunday, March 28, 2021 4:01 AM

1.Synchronized will work fine main

```
//what is the output of the below program:
public class TrickyProgrammingQuestion_1 {

    public static void main(String[] args) {
        test(null);
    }

    public static void test(Object o) {
        System.out.println("Object impl..");
    }

    public static void test(String s) {
        System.out.println("String impl..");
    }

    public static void test(Integer i) {
        System.out.println("Integer impl..");
    }
}
```

```
public class TrickyProgrammingQuestion_1 {

    public static void main(String[] args) {
        test(null);
    }

    public static void test(Object o) {
        System.out.println("Object impl..");
    }

    public static void test(String s) {
        System.out.println("String impl..");
    }

/*  public static void test(Integer i) {
        System.out.println("Integer impl..");
    }*/
}
```

```
String impl..
```

String is more specific

How can we maintain Immutability of a class with a mutable reference ?

1. Make your class ***final,*** so that no other classes can extend it.
2. Make all instance variables **private** & ***final,*** so that they're initialized only once inside the constructor and never modified afterward.
3. Provide only getter methods don't provide setter methods.
4. If the class holds a mutable object:
   - Make sure to always return a clone copy of the field and never return the real object instance.

Do all properties of an Immutable Object need to be final?

```
public final class Employee {

    private  Integer id;
    private final String name;
    private final String email;

    private final Date dob;
    private final Set<String> skills;
    private final Address address;

    public Employee(Integer id, String name, String email, Da
        super();
        this.id = id;
        this.name = name;
        this.email = email;
        this.dob = dob;
        this.skills = skills;
        this.address = address;
    }

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

```
public Date getDob() {
    return new Date(dob.getTime());
}

public Set<String> getSkills() {
    return new LinkedHashSet<>(skills);
}

public Address getAddress() {
    Address empAddress = new Address();
    empAddress.setStreet(address.getStreet());
    empAddress.setZipCode(address.getZipCode());
    empAddress.setAddressLine1(address.getAddressLine1());
    empAddress.setAddressLine2(address.getAddressLine2());
    return empAddress;
}
```

```
     */
    public static final double POSITIVE_INFINITY = 1.0 / 0.0;

    /**
     * A constant holding the negative infinity of type
     * {@code double}. It is equal to the value returned by
     * {@code Double.longBitsToDouble(0xfff0000000000000L)}.
     */
    public static final double NEGATIVE_INFINITY = -1.0 / 0.0;

    /**
     * A constant holding a Not-a-Number (NaN) value of type
     * {@code double}. It is equivalent to the value returned by
     * {@code Double.longBitsToDouble(0x7ff8000000000000L)}.
     */
    public static final double NAN = 0.0d / 0.0;

    /**
     * A constant holding the largest positive finite value of type
```

What is aggregation, How is it different from composition ?

```
public final class Employee {

    private  Integer id;
    private  String name;
    private  String email;
    private  Address address;

    public Employee(Integer id, String name, S
        super();
        this.id = id;
        this.name = name;
        this.email = email;
        this.address = address;
    }
```

Address aggregation-> Address will be there even employee nullify

```
public class Car {
    //final will make sure engine is initialized within C
    private final Engine engine;

    public Car(String engineType){
        engine  = new Engine();
        engine.setType(engineType);
    }

    public Engine getEngine() {
        return engine;
    }
}
```

Aggregation and composition are special type of association and differ only in weight of relationship.

Composition is stronger form of **"is part of"** relationship compared to aggregation **"has a".**

In **composition**, the member object can not exist outside the **enclosing class** while same is not true for **Aggregation**.

Quest 1.What is difference between Stack and Heap area of JVM Memory? What is stored inside a stack and what goes into heap?

Stack

- Memory of **Stack** Section is bound to a method context and is destroyed once a thread returns from the function i.e. the **Stack** objects exists within the scope of the function they are created in.
- Stack section of memory contains methods, local variables and reference variables and all these are cleared when a thread returns from the method call.

Heap

- **Heap** objects exists outside the method scope and are available till **GC** recollects the memory.
- Java stores all objects in Heap wether they are created from within a method or class.
- All class level variables and references are also stored in heap so that they can be accessed from anywhere. Metadata of classes, methods also reside in Heap's PermGen space.

Quest 2. An ArrayList is created inside a method, will it be allocated in Stack section or Heap section of JVM Memory?

```
public void m(){
ArrayList<String> myList = new ArrayList<>();
}
```

#kkjavatutorials #Java #JavaMemoryModel

How Heap space is divided in Java? || What is Metaspace ? || Explain Java Memory model in detail?

- Memory taken up by the JVM is divided into **Stack**, **Heap** and **Non Heap** memory areas.
- **Stacks** are taken up by individual threads for running the method code while heap is used to hold all class instances and arrays created using new operation.
- **Non-heap memory** includes a method area shared among all threads and is logically part of the heap but, depending upon the implementation, a Java VM may not invoke GC on this part.

Why We should never invoke GC programmatically from within your code ? || Garbage collector in java

Invoking **System.gc()** may have significant performance side effects on the application.
GC knows when to invoke **partial or full collection**. Whenever there is a need it tries to collect the space from Young Generation First (very low performance overhead), but when we force our **JVM** to invoke **System.gc(),** JVM will do a **Full GC** which might pause your application for certain amount of time.

Scenario based java garbage collection interview question || Java Online Training

If our young generation is small, then the short lived objects will be promoted to Tenured Generation and thus causing frequent major collection. This can be addressed by setting appropriate value for **-XX:NewSize** parameter at the JVM startup.

Output of program

```
public class B extends A {
    public B() {
        greeting();
        prints();
    }

    void greeting() {
        System.out.println("instance method from B");
    }

    static void prints() {
        System.out.println("Static method from B");
    }
}
```

```
public class A {
    public A() {
        greeting();
        prints();
    }

    void greeting() {
        System.out.println("instance method from A");
    }

    static void prints() {
        System.out.println("Static method from A");
    }
}
```

```
//What is the output of the below program?
public class TrickyProgrammingQuestion {

    public static void main(String[] args) {
        new B();
    }
}
```

A printValue
B greeting (non static)
B printValue
B greeting

When should we choose Array, ArrayList, LinkedList over one another for a given Scenario and Why?

- Array is a fixed size collection which can hold primitive or Objects. Array itself is a object and memory for array object is allocated on the Heap.
- Array does not provide useful collections methods like add(), addAll(), remove, iterator etc.
- We should choose array only when the size of input is fixed and known in advance and underlying elements are of primitive type.

- ArrayList allows Big O(1) time complexity (constant time) for read/update methods. If position of the element is known then it can be grabbed in constant time using get(index) operation.
- Adding or removing elements from ArrayList (other than at end) requires shifting elements, either to make a new space for the element or for filling up the gap. Thus if frequent insertions and removals are required by your application logic then ArrayList will perform poorly (roughly Linear Time Big O(n)).
- The size, isEmpty, get, set, iterator, and listIterator operations run in constant time.
- If more elements are needed than the capacity of the ArrayList then a new underlying array with twice the capacity is created and the old array is copied to the new one which is time consuming operation (roughly Big O(n)).
- To avoid higher cost of resizing operation, we should always assign a appropriate initial capacity to the ArrayList at the time of construction.

- LinkedList provides constant-time (Big O(1)) methods for insertion and removal using Iterators. But the methods to find the elements have Big O(n) time complexity (Linear Time, proportional to the size of list) and thus are poor performing.
- LinkedList has more memory overhead because it needs two nodes for each element which point to previous and next element in the LinkedList. If you are looking for random access of elements then ArrayList is the way to go for.

What all collections utilizes hashCode method? ||Hashing Data Structure in Java Interview Questions?

- Only hashing data structures uses **hashCode()** method along with equals() method, though the equals() is used by almost every class.
- hashCode is useful for creating hashing based datastructures like **HashMap, LinkedHashMap, ConcurrentHashMap, HashSet.** (Basically any Java collection that has Hash inside the name of it)
- Hashcode is used to provide best case O(1) time complexity for searching the stored element.
- **TreeMap, TreeSet** uses Comparator/Comparable for comparing the elements against each other, so these data structures do not require hashCode() method. The best case time complexity offered by these datastructures for lookup operation is logarithmic rather than constant.

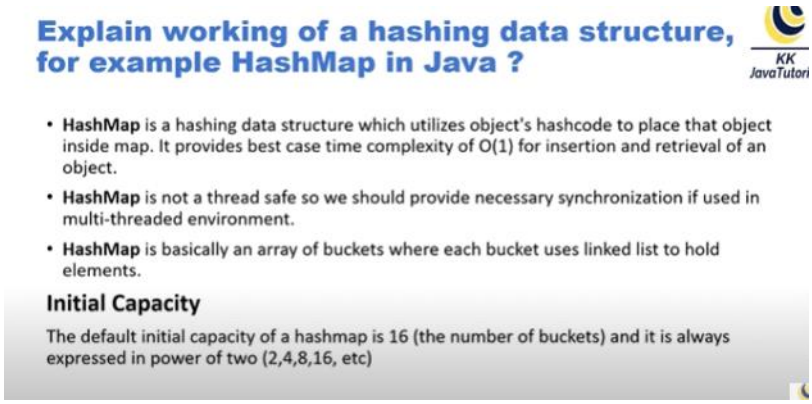
Load factor and Rehashing

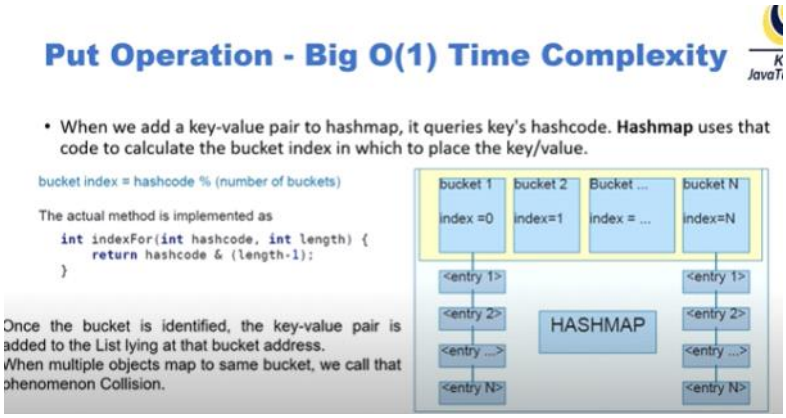
KK JavaTut

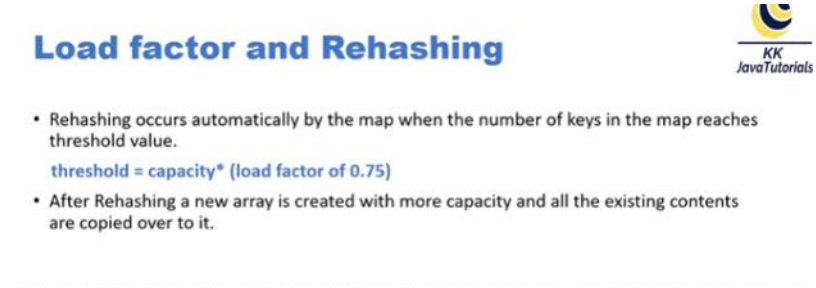
- Rehashing occurs automatically by the map when the number of keys in the map reaches threshold value.

  threshold = capacity* (load factor of 0.75)

- After Rehashing a new array is created with more capacity and all the existing contents are copied over to it.





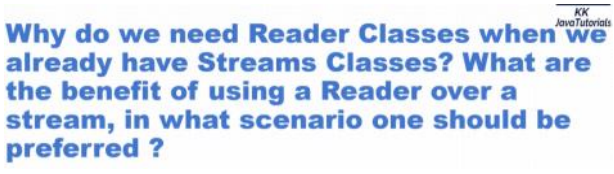


There will be buckets of linkedlist. In which key will be hashed. If key is present then it will update the value. If it is not present it will just put at the end of bucket.

Number of value/No of buckets-> Loading factor-> Average element of buckets. 0(Lamabada). It is compared in that bucket only.
Lambada should be less than thresold value.

We have to rehash

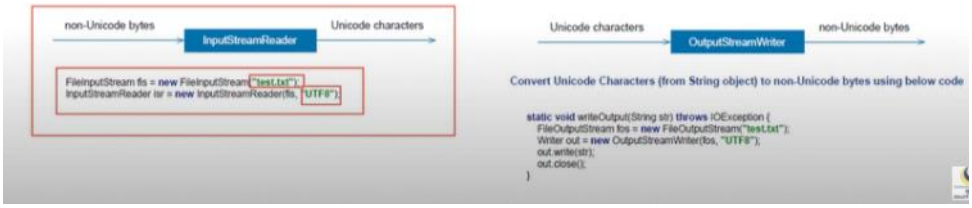
Get-> Just search in that bucket. If it is found then return true otherwise false;



- **Byte Streams** – These handle data in bytes (8 bits) i.e., the byte stream classes read/write data of 8 bits. Using these you can store characters, videos, audios, images etc.
- **Character Streams** – These handle data in 16 bit Unicode. Using these you can read and write text data only.

From <https://www.tutorialspoint.com/difference-between-the-byte-stream-and-character-stream-classes-in-java>

- It is possible to convert byte stream to a character stream using **InputStreamReader** and **OutputStreamWriter**.



How does an ArrayList expands itself when its maximum capacity is reached ? || Java Online Training

From <https://www.youtube.com/watch?v=J8TjKaGlELg&list=PLzS3AYzXBoj84LfxRFAuM-eDlPRWvhGeJ&index=31>

Arraylist define some default capacity when you go beyond that then arraylist increase its size

If hashcode() methodalways returns 0 then what will be the impact on the functionality of software ?

From <https://www.youtube.com/watch?v=Lfz9Ud2n1f0&list=PLzS3AYzXBoj84LfxRFAuM-eDlPRWvhGeJ&index=32>

- **Hashcode** is used to fairly distribute elements inside a map into individual buckets.
- If the hashcode returned is zero for each element then the distribution will no more be fair and all the elements will end up into a single bucket.
- Each bucket in a **HashMap** contains **LinkedList** of HashEntry objects, so in a way **HashMap** will act as a map with single bucket holding all of its elements in a list.

So time complexity of get and put method will become : Big O(n) instead of Big O(1)
Although, functionally it will still behave correctly.

If we don't override hashcode() while using a object in hashing collection, what will be the impact?

From <https://www.youtube.com/watch?v=wKN6G9vZ9tI&list=PLzS3AYzXBoj84LfxRFAuM-eDlPRWvhGeJ&index=33>

- Object's default **hashcode()** method will be used to calculate the hashcode, which in turn will return the memory address of the object in hexadecimal format.
- So in a way the **HashMap** will behave like an identity **HashMap** which will consider two elements equal if and only if two objects are same as per their memory address (and not logically).

```
public class ClientTest {

    public static void main(String[] args) {
        Map<Employee,Department> empDeptMap = new HashMap<>();
        //Map<Employee,Department> empDeptMap = new IdentityHashMap<>();
        empDeptMap.put(new Employee(1001, "Sean", "sean@kkjavatutorials.com"), new Department(1111, "IT", "Delhi"));
        empDeptMap.put(new Employee(1002, "John", "john@kkjavatutorials.com"), new Department(2222, "IT", "Mumbai"));
        empDeptMap.put(new Employee(1003, "Sam", "sam@kkjavatutorials.com"), new Department(3333, "Finance", "Chennai"));
        empDeptMap.put(new Employee(1001, "Sean", "sean@kkjavatutorials.com"), new Department(1112, "IT", "Delhi"));


        empDeptMap.forEach((k,v)->System.out.println(k+"\t"+v));
    }
```

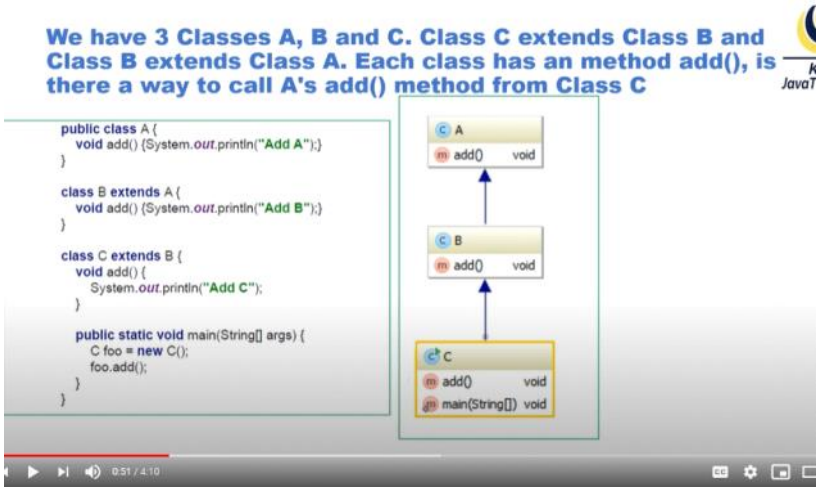
```
<terminated> ClientTest (1) [Java Application] C:\Program Files\Java\jre1.8.0_65\bin\javaw.exe (02-Jun-2019, 10:22:02 am)
Employee [id=1001, name=Sean, email=sean@kkjavatutorials.com]   Department [id=1112, name=IT, location=Delhi]
Employee [id=1001, name=Sean, email=sean@kkjavatutorials.com]   Department [id=1111, name=IT, location=Delhi]
Employee [id=1002, name=John, email=john@kkjavatutorials.com]   Department [id=2222, name=IT, location=Mumbai]
Employee [id=1003, name=Sam, email=sam@kkjavatutorials.com]     Department [id=3333, name=Finance, location=Chennai]
```

```
public class ClientTest {

    public static void main(String[] args) {
        Map<Employee,Department> empDeptMap = new HashMap<>();
        //Map<Employee,Department> empDeptMap = new IdentityHashMap<>();
        empDeptMap.put(new Employee(1001, "Sean", "sean@kkjavatutorials.com"), new Department(1111, "IT", "Delhi"));
        empDeptMap.put(new Employee(1002, "John", "john@kkjavatutorials.com"), new Department(2222, "IT", "Mumbai"));
        empDeptMap.put(new Employee(1003, "Sam", "sam@kkjavatutorials.com"), new Department(3333, "Finance", "Chennai"));
        empDeptMap.put(new Employee(1001, "Sean", "sean@kkjavatutorials.com"), new Department(1112, "IT", "Delhi"));


        empDeptMap.forEach((k,v)->System.out.println(k+"\t"+v));
    }
```

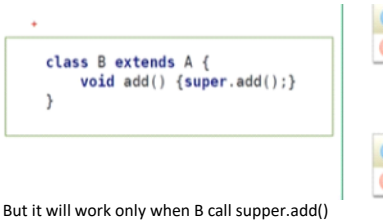
```
Employee [id=1001, name=Sean, email=sean@kkjavatutorials.com]   Department [id=1112, name=IT, location=Delhi]
Employee [id=1002, name=John, email=john@kkjavatutorials.com]   Department [id=2222, name=IT, location=Mumbai]
Employee [id=1003, name=Sam, email=sam@kkjavatutorials.com]     Department [id=3333, name=Finance, location=Chennai]
```

After uncommenting hashcode

Identity hashmap will print 4 hashcode will remove overide

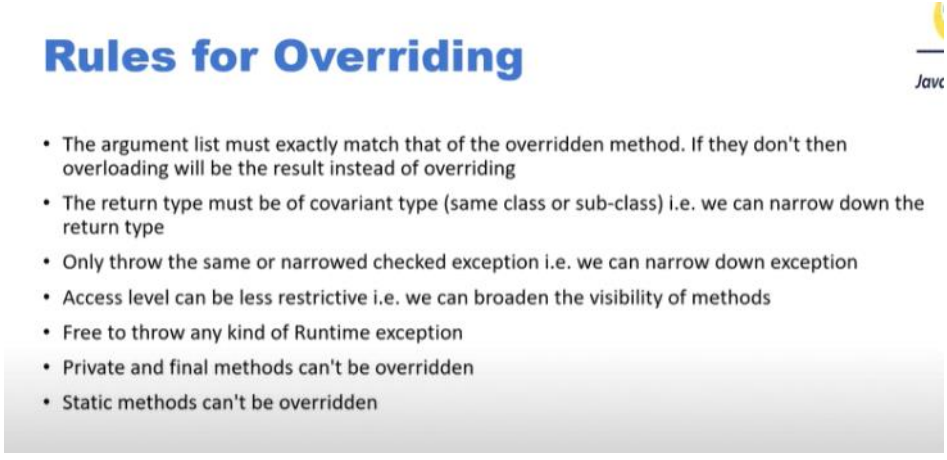


C can never see class a method. It will violate oops concept



```
class B extends A {
    void add() {super.add();}
}
```

But it will work only when B call supper.add()



```
public static void main(String[] args) {
    ArrayList<Integer> arrayList = new ArrayList<Integer>();
    arrayList.add(100);
    arrayList.add(20);
    arrayList.add(40);
    arrayList.add(60);
    arrayList.add(35);
    System.out.println("Original List::"+arrayList);
    deleteFromCollection(arrayList);
    System.out.println("After Removing elements From List::"+arrayList);
}

public static void deleteFromCollection(List<Integer> marks) {
    marks.removeIf(i->i<40);//Will not throw java.util.ConourrentModificationException
}
```

Iterator interface did not have add() method. What could be the reason for such behavior?

From <https://www.youtube.com/watch?v=B83vH18c7wA&list=PLzS3AYzXBoj84LfxRFAuM-eDlPRWvhGeJ&index=40>

ListIterator provide add method because it know where to add the value. Because it is in order
Because iterator doent know underlying collection . It can be set or list

Can the keys in Hashing data structure be made Mutable ?

- The answer is **NO**. If we make the keys mutable then the **hashcode()** of the key will no more be consistent over time which will cause lookup failure for that object from the data structure.

```
public static void main(String[] args) {
    Map<Employee,Department> empDeptMap = new HashMap<>();
    Employee employee = new Employee(1001, "Sean", "sean@kkjavatutorials.com");
    Department department = new Department(1111, "IT", "Delhi");
    empDeptMap.put(employee, department);

    Department department1 = empDeptMap.get(employee);
    System.out.println("Before changing key = "+department1);

    employee.setName("Sean Murphy");//Modifying Key

    System.out.println("------------------------");
    Department department2 = empDeptMap.get(employee);

    System.out.println("After changing key = "+department2);

}
}
```

```
ore changing key = Department [id=1111, name=IT, location=Delhi]
----------------------
er changing key = null
```

Guidelines to avoid Deadlock

KK JavaTuto

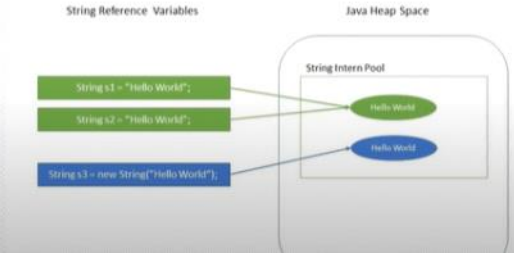
1. Avoid acquiring multiple locks at once, if it is absolutely required then always acquire the lock in the same order and release them in same order across the multiple methods/threads.
2. Avoid calling un-trusted foreign code while holding a lock. Time consuming calls should be avoided from within the locks.
3. Use timed & interruptible locks i.e. put a timeout on the lock attempt, if a thread is unable to acquire the lock within the given timeout value then it should free up all the acquired locks and retry after sometime. Java provides **Lock** Interface for this specific purpose.
4. Avoid locks using lock-free data-structures like **ConcurrentLinkedQueue** instead of a synchronized **ArrayList.**
   Use **ConcurrentHashMap** instead of synchronized **Hashtable**. Java provides many lock-free APIs in its atomic package like **AtomicInteger,AtomicLong** etc.

- Immutable objects are inherently thread-safe
- It helps to writing multi-threading code without much worries.
- Immutable objects are good candidate for hash keys because their hashcode can be cached and reused for better performance

Why shouldn't we prefer mutable static variables in our Java Code?

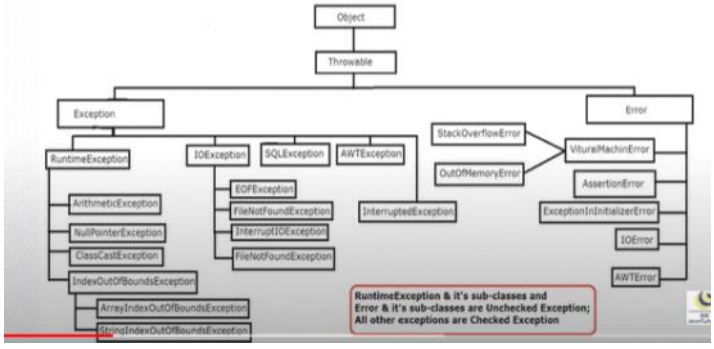
- Using mutable static variables might introduce Bugs in your software at some point in time.
- Problem sharing a mutable static variable in multi-threaded environment. It's very tough to write & maintain a thread safe code with Mutable non-private static fields.
- Code that relies on static objects can't be easily unit tested, and statics can't be easily mocked and hence does not promote TDD.

- JVM has a string pool where it keeps at most one object of any String. String literals objects are always created in the string pool for reusability purpose. String objects created with the new operator do not refer to objects in the string pool.



- Immutability brings inherent thread-safety to the usage of String class in a concurrent program. So multiple threads can work on the same String object without any data corruption. There is absolutely no need to synchronize the code because of String objects.
- StringPooling is possible only because of immutability because the underlying contents of the String will never change. This helps reducing the overall memory footprint of the application using lots of String objects.
- Hash code for Immutable objects are calculated lazily on first usage and then cached for future reference. This gives the benefit of performance when we use Immutable Key's in any hashing data structure.

ptions



HashMap

HashMap is a hashing data structure which works on **hashcode** of keys. Keys must provide consistent implementation of equals() and hashCode() method in order to work with hashmap. Time complexity for get() and put() operations is Big O(1).

TreeMap

TreeMap is a SortedMap, based on Red-Black Binary Search Tree which maintains order of its elements based on given comparator or comparable. Time complexity for put() and get() operation is O (log n).

LinkedHashMap

LinkedHashMap is also a hashing data structure similar to HashMap, but it retains the original order of insertion for its elements using a LinkedList. Thus iteration order of its elements is same as the insertion order for **LinkedHashMap** which is not the case for other two Map classes. Iterating over its elements is lightly faster than the HashMap because it does not need to traverse over buckets which are empty.

In Java, every object has a built in lock (or monitor) that gets activated when Object has synchronized method code. It is basically of two types

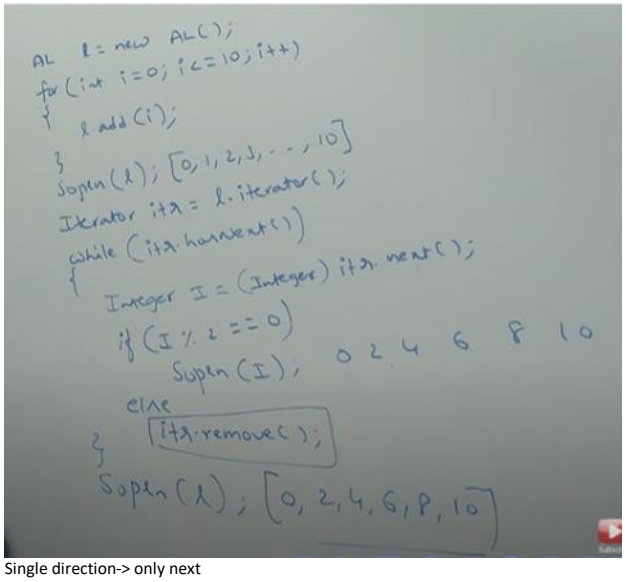
- **Instance Lock**
- **Class Lock (Locking on static methods or on Class object)**

- When we enter a non-static synchronized code then JVM acquires instance level lock on the Object whose method we are executing. Instance lock can also we acquired using the following syntax with block level synchronization.

Notes about instance locking

- Lock is mutually exclusive, which means that only one thread can acquire it and other threads have to wait for their turn until first thread releases it.
- Each Java object has just one lock (or monitor)
- Non-synchronized methods and a single static synchronized method can be executed in parallel.

Instance -> Object level
Static locking-> class level lock



Single direction-> only next