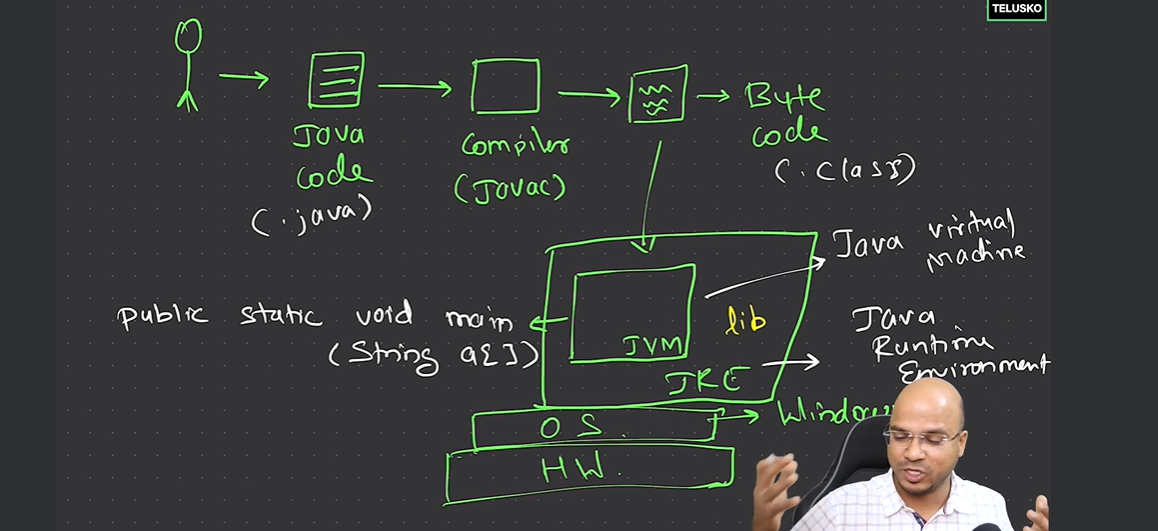
Udemy java developer course learnings :

**CORE JAVA**

**DAY 1 – April 06**

* Compiler will convert any type of code to byte code.
* Jre + JIT convert byte code to machine code.
* Java version check by using –> java --version
* Java compiler version check by using -> javac –version.
* **Jshell** was introduced in java9.
* By default every OS will have JVM installed.
* Java is platform independent : runs on any OS like windows, linux, mac etc…
* WORA – write once run anywhere . JDK is not required as we are having byte code withus and by default every machine will have JRE which will have JVM.
* Java – run application with .class extension.
* Javac – compile the java application.
* How java application runs ?



* Java is called **strongly typed language**.  
  A strongly typed programming language is one in which each type of data, such as integers, characters, hexadecimals and packed decimals, is predefined as part of the programming language, and all constants **or variables defined for a given program must be described with one of the data types.**
* Data types in java – primitive and Non-primitive(wrapper).
* Primitive data type – Integer,float,character,Boolean
* Types in Integer – byte,short,long,int
* Types in float – double,float
* 1 byte = 8 bits
* Integer – **size** -> 4 bytes, **range** -> -231 to 231-1(default)
* Long– **size** -> 8 bytes, **range** -> -263 to 263-1
* short– **size** -> 2 bytes, **range** -> -215 to 215-1
* byte – **size** -> 1 bytes, **range** -> -27 to 27-1
* float – **size** -> 4 bytes.
* Double – **size** -> 8 bytes (**default**).
* Boolean – size -> 1 byte - false (default)
* If you declare float num = 5.6 then it will throw error since it will consider it as double by default, so to overcome this we need to declare float num = 5.6f; lly for long we need to declare as long num = 58884l or long num = 58884L;
* Char – 2 bytes
* 1 byte = 8 bits.
* **Literal** - eg:8,855   
  Literals in Java are a synthetic representation of boolean, character, numeric, or string data. They are a means of expressing particular values within a program. **They are constant values that directly appear in a program and can be assigned now to a variable.**
* **binary format** -> int num = 0b101; prints 5
* **Hex** -> int num = 0x7E prints 126(A=10,B=11,C=12,D=13,E=14,F=15 so num = (7x161) + (14x160) = 126.
* To differentiate between no of zeros java supports -> int num1 = 10\_00\_00\_000; prints 100000000.
* Double num1 = 12e10; -> e represents **eplision** here which it means 1010 .
* Character can also be treated as int .   
  eg: char num = ‘a’;  
   num++;   
  printing num will prints b, char c = 97 is also a valid declaration.
* The range of **ASCII values** for uppercase letters **A-Z** is 65-90, and the range for lowercase letters **a-z** is 97-122.
* **Type casting** : eg : byte b =125; int a = 12; b = (byte) a;
* **Type conversion** eg: a = b;
* In type casting if the value goes beyond range then it will use modulus. Eg : byte b = 127; int a = 257; b = (byte) a; printing b will prints 1 . As the range of byte is -128 to 127 in total 256, so it will do 257%256 prints 1;  
  if the int values is in between -128 to 127 then byte will take it as such.  
  if the int values exceeds 127 then it consider it as 128-256 = -128  
  and also for -129 it consider it as 256-127=127
* byte b = 127; int a = 250; b = (byte) a; printing b gave answer as -6 as 250-256=-6.
* **Type promotion** : eg : byte a=10; byte b=20; int result = a\*b; printing result will give 200, here 200 is out of range of byte.

**DAY 2 : April 07**

* OR operator (||) is called **short circuit**. If first condition satisfies it won’t check for second condition. Same is the case for && -> if first condition fails it won’t check for second condition.
* Syntax of do while  
  int i =5;  
  do{  
   System.out.println(“Hi” +i);  
   i++;  
   }while(i<=4);  
  **output**: Hi 5

**Mistakes made in mcq’s**

**1**. What is the output of the following program in Java?

if(1) {

System.out.println("Navin Reddy");

}

**Output** : compile-time error as int cannot be converted to boolean.

**2**. What is the type of the do-while loop in terms of control flow? – **Exit Controlled loop**.

**Exit Controlled loop** : An exit-controlled loop is a type of Loop in computer programming that tests the loop condition at the **end of the Loop** after executing the body of the Loop at least once. As the program flow reaches an exit-controlled loop, the loop body is executed first, and then the loop condition is tested.

**Entry Controlled loop** : condition is checked before the body is executed.

Eg: if, for, while.

**3**. What will be the output of the given code snippet?

public static void main(String[] args) {

int i, j;

i = 100;

j = 300;

while(++i < --j);

System.out.println(i);

}

**Output** : 200

**DAY 3 : April 08**

* JVM creates objects in java.
* Stack will store data in key value pairs.
* Object will be created inside heap memory and the object address is pointed in stack.
* By default all values are set to zero in array in which size is mentioned.
* **Math.random()** will generate random values of **double** datatype less than 1.
* **Main()** method can be **overloaded** but it will consider the one which is default now.
* **Main() method cannot be overridden** because it is static method. Static methods can not be overridden, as they are associated with class and while non-static methods are associated with objects. Static methods can be accessed using className.methodName().

**MCQ’s mistakes**

1.what will happened if a local variable has the same name as an instance variable within a class ? Ans: Local variable hides the instance variable (which means local variable takes preference over instance variable).

2.Instance variables memory allocated in – **heap**.

3.Which subsystem of the JVM is responsible for loading class files ? **ClassLoader**.

**What is classLoader ?  
A ClassLoader in Java is a part of the Java Runtime Environment (JRE) that is responsible for dynamically loading classes into the Java Virtual Machine (JVM) at runtime.** It loads the bytecode of a class into memory when the class is referenced for the first time in the program.

**Types of Class Loaders in Java**

Java provides a hierarchy of class loaders:

1. **Bootstrap ClassLoader:**
   * The parent of all class loaders.
   * Loads the core Java classes from the Java Development Kit (JDK), such as classes in java.lang, java.util, etc.
   * Implemented in native code, and it doesn't have a parent.
2. **Extension ClassLoader:**
   * Loads classes from the java.ext.dirs directory (e.g., lib/ext folder in the JDK).
   * It loads the extensions of the standard Java classes.
3. **Application (or System) ClassLoader:**
   * Loads classes from the application's classpath (e.g., CLASSPATH environment variable, -cp or -classpath option).
   * Typically used to load application-level classes.

**Working of Class Loaders**

When a class is needed, the JVM delegates the class loading task to the appropriate ClassLoader.

Java uses a parent-delegation model for class loading:

A class loader first delegates the loading request to its parent.

If the parent can't find or load the class, the current class loader attempts to load it.

**public class ClassLoaderExample {**

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

**// Get the class loader of this class**

**ClassLoader classLoader = ClassLoaderExample.class.getClassLoader();**

**// Print details of the class loader hierarchy**

**System.out.println("ClassLoader of this class: " + classLoader);**

**System.out.println("Parent ClassLoader: " + classLoader.getParent());**

**System.out.println("Grandparent ClassLoader: " + classLoader.getParent().getParent()); // This might return null**

**}**

**}**

**Output:**

ClassLoader of this class: jdk.internal.loader.ClassLoaders$AppClassLoader@1b6d3586

Parent ClassLoader: jdk.internal.loader.ClassLoaders$PlatformClassLoader@4554617c

Grandparent ClassLoader: null

**Custom Class Loaders**

You can create your own class loader by extending the ClassLoader class. Custom class loaders are often used in frameworks or tools like web servers (e.g., Tomcat) to dynamically load or reload classes at runtime.

Eg:

public class CustomClassLoader extends ClassLoader {

@Override

protected Class<?> findClass(String name) throws ClassNotFoundException {

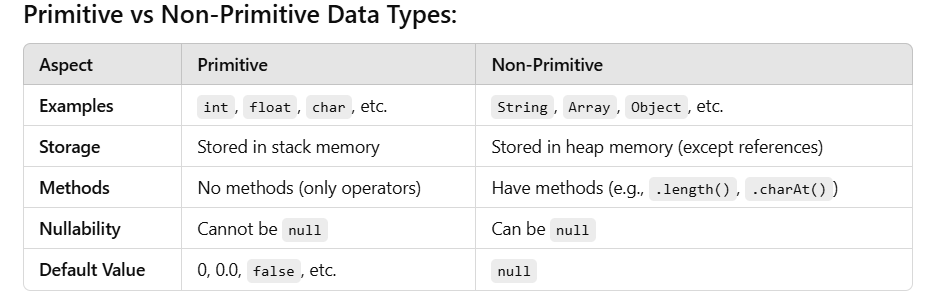
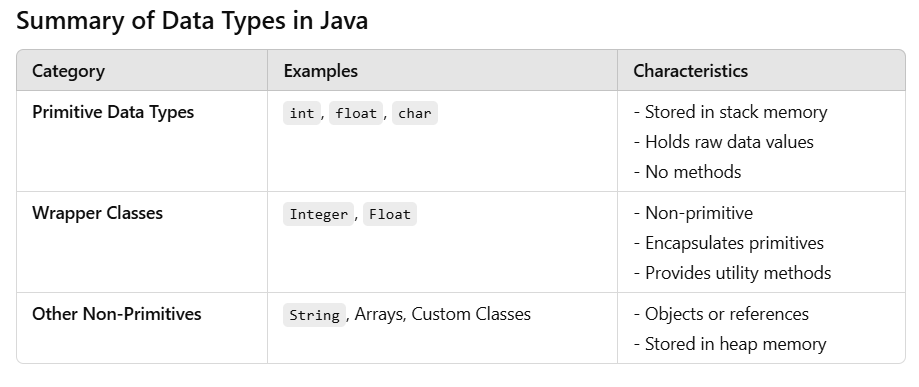
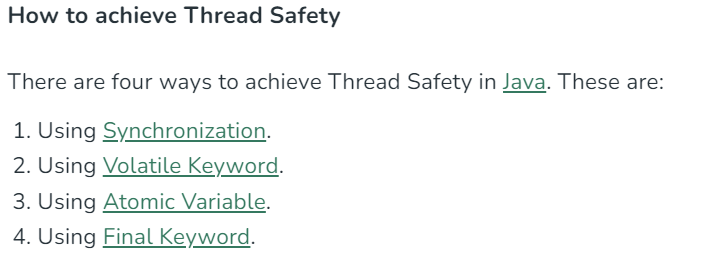
// Custom implementation to find and load the class

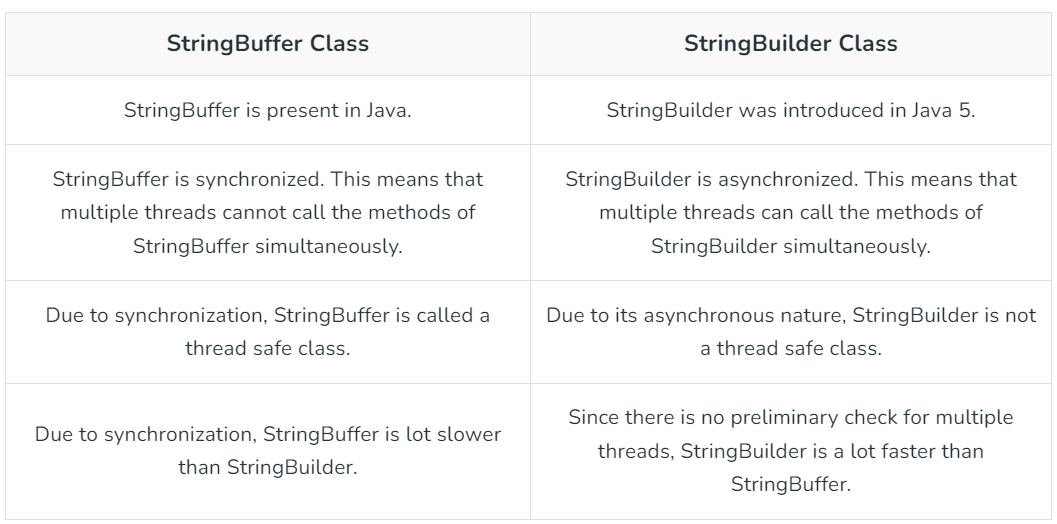
return super.findClass(name);

}

}

**DAY 4 : April 09**

* **Jagged array** – a two dimensional array where we don’t specify no of columns.  
  eg: int nums[] = new int[3][4];  
  we need to specify each row contains no.of columns like this.  
  nums[0]=new int[3]; -- 3 columns  
  nums[1] = new int[4];--4 columns  
  nums[2]=new int[2]; -- 2 columns  
  assign random values using Math.random() like this  
  for(int i=0;i<nums.length;i++){  
  for(int j=0;j<nums[i].length;j++){  
  nums[i][j] = (int) (Math.random()\*10);  
  }  
  }  
  output: 7 5 8   
   2 9 3 1  
   3 9
* Similar to 2D array we are having 3D array as well.
* Drawbacks of array:  
  1.length is fixed.  
  2.for any operation like searching, you need to traverse entire and takes lot of time.  
  3.can’t store different datatypes at a time.
* Exceptions are runtime errors.
* forEach loop is also called enhanced for loop.
* **String** is not a primitive data type, it is a **class** so belongs to non primitive.
* 
* 
* To find hash code in string – **stringName.hashCode();**
* **SCP**(String Constant Pool) is present inside **heap**.
* String s1 = “Hari”;  
  String s2 = “Hari”;  
  Actually here it needs to create 2 objects but internally it will create one and both will points to same in SCP. To check this we can use s1==s2.  
  the hash code is also same for both s1 and s2.
* Types of Strings : mutable (changeable) and immutable (unchangeable).  
  **immutable** : eg : String  
  **mutable** : eg: StringBuffer and StringBuilder
* String buffer is **thread safe** where as string builder is not thread safe.
* **What is thread safe means** ?  
  In multithreaded environments, we need to write implementations in a thread-safe way. This means that different threads can access the same resources without producing unpredictable results.
* A **volatile** keyword is a field modifier that ensures that the object can be used by multiple threads at the same time without having any problem.
* How to achieve thread safety in java ?  
  https://www.geeksforgeeks.org/thread-safety-and-how-to-achieve-it-in-java/  
  
* Difference between String buffer and String builder ?



* If you want a property whose value needs to be **same for all the objects** then you need to declare it as **static**.It will remain same for all. If you change it with className.propertyName or object.propertyName then at once it will effected in other objects as well.
* Initialization of a static variable is not mandatory. Its default value is dependent on the data type of variable. For String it is null, for float it is 0.0f, for int it is 0, for Wrapper classes like Integer it is null, etc.
* Static variables needs to be called with class Name.(you can also call with obj , no problem in that).
* **Non static methods can use static variables**.
* Static methods can be called with class name. .(you can also call with obj , no problem in that).
* **Inside static method you can only use static variables**.If you want to use nonstatic variables then you need to have an object as reference to static method then use it.
* **Why we have static in public static void main (String args[]) ?**  
  if we don’t have static , then to access main method we need to create object of the class and call the main method. Since we have it as static we don’t need to create the object of it , instead we can directly call it with class name itself.
* **Static block** – for assigning any value. The static block can be called **only once** irrespective of how many objects you created.  
  eg: static {  
   name = “phone”;  
  }  
  here name is a static variable.
* **Using static block we can print in the console with out main method**.
* The static block is called **before** the constructor.
* Class is loaded first in **class loader** then objects are loaded. Static block is called inside class loader so it is called once, where are constructor is called for each object.
* If **no object is created** then class loader will not load the class, so static method itself will not be called. To load the class in this scenario use **Class.forName(“Mobile”);** where Mobile is the class name. So only the static method alone will be called not the default constructor.

**DAY 5 : April 11**

* Variable declared as private inside the class, cannot be directly accessed with class object. Instead you need to have methods(getters and setters) inside class to access/modify it. This process of binding variables with methods is called **encapsulation**.
* **this** refers to current object who is calling the method.
* Default values for integer and string are 0 and null inside a class and trying to access via object. For all nonprimitive data type (Wrapper class) the default value is null.
* Objects without name is called **anonymous object.**  
  **eg**: if there exists a class A() and one more class B() which contains main method.  
  inside of B() if you call want to call A() as anonymous then you need to call like this.  
  **new A();**  
  **disadv**: cannot reuse (because it is not initialized to any variable)
* **Redundancy** : repeating same code multiple times, which is a bad practice.
* Class name and interface name should start with capital letters.  
  eg: Human  
  variable names should follow camel casing for better readability.  
  eg: isMyShow  
  constant should always be caps.  
  eg: VALUE, MAX\_VALUE
* **Multiple inheritance** in java will not work.  
  eg: class C extends A,B // throws error because of ambiguity error if same method is present in both A,B which C is trying to access.
* If we create an object for subclass then it will call default constructor for both sub class and super class.

class A{

A(){

System.out.println("Inside class A constructor");

}

}

class B extends A{

B(){

System.out.println("Inside class B constructor");

}

}

class GFG {

public static void main (String[] args) {

B obj = new B();

}

}  
**output** :   
Inside class A constructor

Inside class B constructor

* By default in every constructor the first statement is super().(both default constructor and parameterized constructor).
* If you explicitly call parameterized constructor of super class from child class default constructor using super(parameter) then default constructor of super class will not get executed. Only constructor with parameter of super class and default constructor of child class gets executed.
* Every class in java **by default extends object class**.
* this(); - will execute the constructor of same class (you can use it inside parameterized constructor for testing).
* By default **every class belong to a package** in java.
* By **default java.lang.\*; is imported in all files** which is not visible.

**DAY 6 : April 12**

* If two classes are in same package and one class is having a variable defined as int marks=6 then it can be accesed in other class.
* If two classes are in different package and one class is having a variable defined as int marks=6 and trying to access in other class then it will throw error, to make it accessible you need to change it to public int marks = 6.
* If you want something **to access outside the package then declare it as public.**
* **Variable declared as private should be used in same class**.
* **If none of the access modifier specified then it will take default**.
* **Default – it can be accessed in the same package**.
* **Protected – can be used in same class, same package ,in different package as subclass**.
* **Polymorphism** :  
  types:

1.Compile time (or) Early binding – eg: Overloading  
2.Runtime (or) LateBinding – eg: Overriding

* **Dynamic method dispatch** – term related to overriding.  
  Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.  
  way to resolve overridden method calls at run time instead of compile time.
* **Variable declared as final ,its value cannot be changed.**  
  Making a **class as final**, no other classes **cannot be able to extend it**.  
  Making a **method as final**, **no one can override**.
* When ever you are trying **print object, by default obj.toString()** will be printed.

Eg: Laptop@7ad041f3  
pattern : className + @ + hex(hashcode)

* **.equals() on objects compare the objects based on hexadecimal values** not based on content.
* The reason why equals() and toString() needs to override is understood now. The reason is if we don’t overridden, then the methods present in object class is taken into consideration.
* For string values comparison use equals() and for numeric value comparison use ==
* Consider there are two classes A and B where B extends A, A has method show1() and B has method show2()  
  **upcasting** : A obj = (A) new B(); //same as new B()   
  using obj we can only access show1() not show2() since obj is reference of A.  
  **downCasting** : B obj1 = (B) obj;  
  here we can able to access show2() as we can casted the A to B type.
* Java is not 100% object oriented because of primitive datatypes. **Collections** only work with classes in java, so to work on in collections we need to use wrapper class.  
  int -> Integer  
  char -> Character  
  double -> Double
* int num=7;  
  Integer num1 = new Integer(num); // -- this concept is called **boxing**.  
  Integer num2 = num; // **autoboxing**.  
  int num3 = num1.intValue(); // **unboxing**  
  int num4 = num1; // **auto-unboxing**;  
  parseInt() is also a feature of **wrapper class**.

1st Revision of core java done on April 13.

2nd Revision of core java done on April 20.

3rd Revision of core java done on April 28.

4th Revision done on Jun 2.

5th Revision done on Jun 26.

Interview topics are

Java8 features all, latest version of java and what are its features, collections related questions , String buffer string builder, threads concepts.

**ADVANCED JAVA**

**April 13, 2024**

* Taking input from user using Scanner.  
  Scanner sc = new Scanner(System.in);  
  int res = sc.nextLine();  
  Scanner class is present in java.util.Scanner;
* Abstract methods can only be present inside abstract class.
* If you are not sure of a method implementation then you can declare that method as abstract and you can define that method inside the subclass. The subclass should definitely override the abstract method present in the parent class.
* You can not create an object of abstract class but you can create a reference of it w.r.t subclass it is has defined all the methods.
* It is not mandatory that abstract class should contains abstract methods.
* If you can’t override the abstract method inside child class then you need to make the child class as abstract.
* The class in which where all abstract methods are defined and we are able to create objects is called **concrete class.**
* Abstract class having methods other than abstract methods are called **concrete methods.**
* Class inside a class is called **innerclass**.
* Seen **anonymous inner class**, where the method can be defined while creating object.

**Eg:**

class A

{

public void show()

{

System.out.println("in A show");

}

}

public class AnonymousInnerClass{

public static void main(String[] args) {

//A obj=new B();

A obj=new A()

{

public void show()

{

System.out.println("in new show");

}

};

obj.show();

}

}

* Seen **Abstract anonymous inner class**, where the abstract method is defined while creating object of the abstract class.  
  abstract class A

{

public abstract void show();

public abstract void config();

}  
public class AbstractAnonymousInnerClass{

public static void main(String[] args) {

A obj=new A()

{

public void show()

{

System.out.println("in new show");

}

public void config()

{

System.out.println("in config method");

}

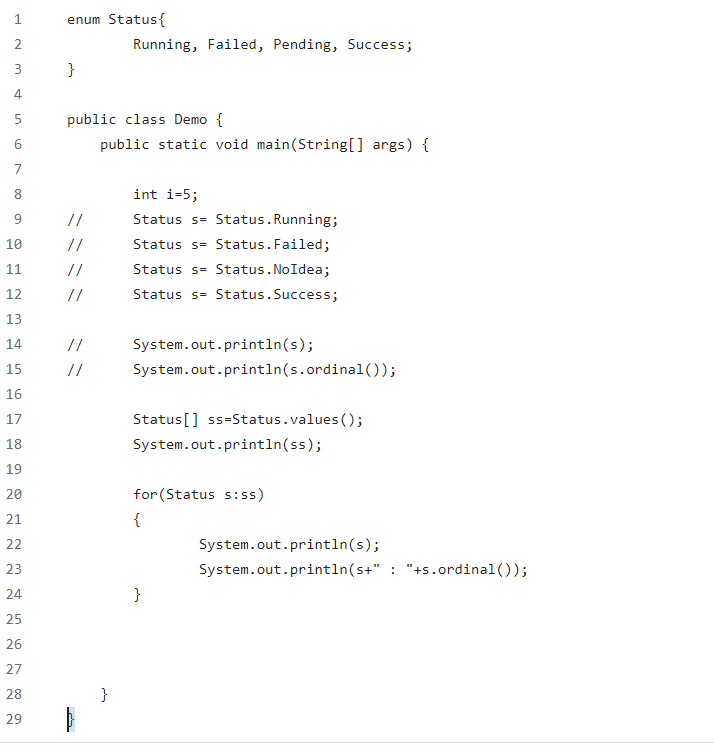
};

obj.show();

}

}  
**April 14, 2024**

* Every method in the **interface** is by default is **public abstract**.
* **Variables** defined in the **interface** are by default **final and static.**
* Class -> class = extends  
  class -> interface = implements  
  interface -> interface = extends
* For named constants prefer using **enum**.  
  eg:



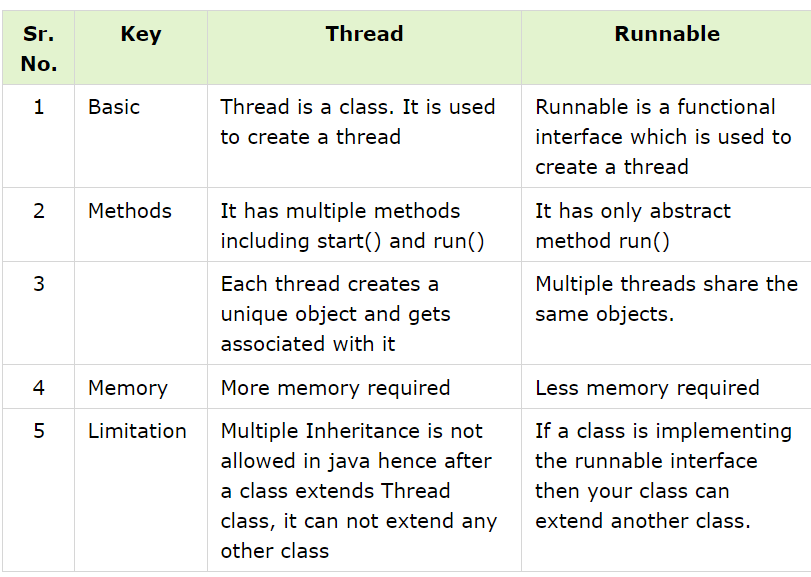
Ordinal() prints the index of it, eg : for running it is zero.

Values will fetch all values in the enum.

* Types of interfaces:  
  1.**Normal Interface** – two or more methods.  
  2.**Functional interface** (**SAM**-Single Abstract Method) – one method.  
  functional interface can have any number of default and static methods.  
  eg : **Comparable,Runnable**  
  3.**Marker Interface** – no methods. Eg: **serializable, Cloneable**Marker interface can have default methods.
* **Lambda expression** can be used only with functional interface. Why ? since in functional interface we only have one method it will understood that the definition which we are writing is only for one method.If we have two methods then it will not understood for which method we are writing lamba function.
* Exceptions:  
  Types of Errors :  
  1.Compile time errors.  
  2.Run time errors - **exceptions**  
  3.Logical errors.
* All runtime exceptions are unchecked exception. Eg: arithmetic exception,ArrayIndexOutofbounds exception,Null pointer exception etc..  
  checked exceptions : SQL exception, IO exception, FileNotFoundException.
* Throw keyword is used to throw exception.  
  eg: throw new ArithmeticException();
* Seen how to implement custom exception.
* e.printStackTrace() will prints entire error stack so that we can check which method is calling which method to handle exception.

**April 15, 2024**

* println method belongs to which class ? - **print stream**
* ways to take input from user in java – **buffer reader, scanner class**
* **try with catch is not mandatory, you can use try with finally also.**
* Seen try with resources demo.
* Thread – smallest unit you can work with. You can run multiple threads. Threads can share resources.
* **Scheduler** is present in the system which will allocate time for the threads to executes. octa core can execute 8 threads at a same time.
* Seen demo on threads, where we extends thread, used start and run. – Multiple Threads video
* Seen demo on thread priority and sleep.  
  thread priority range : 1 to 10 where 1 is least priority and 10 is highest priority.  
  5 is normal priority and by default every thread is having normal priority.
* threadclass extends Runnable interface.
* Runnable VS Thread seen demo need to make differences in table.

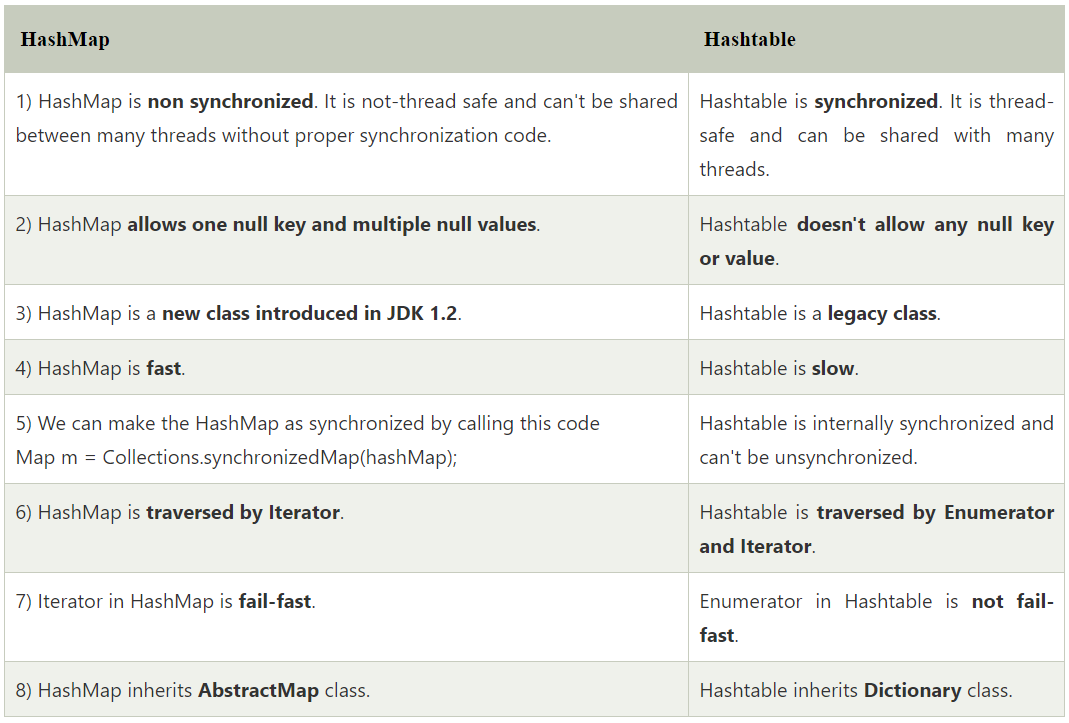


* Start method belongs to thread class and run method belongs to runnable interface.
* We can create reference of a interface object of a class. Eg : Runnable obj1 = new A();
* **Race Conditions** and demo on it. (seen demo where two threads are trying to increment the same variable)
* **Race condition occurs** when multiple threads read and write the same variable i.e. they have access to some shared data and they try to change it at the same time.
* Mutation = changeable.
* **Thread safe** : only one thread will work with a variable at a time.
* **Join** is a method where main method waits for you threads to come back and join.
* **Synchronized** – if two threads are working on same method, then by using synchronized keyword on method which the threads are trying to access, if one thread is using then other thread has to wait until its completion. Saves us from **race** condition. – see demo on race condition.
* **States in thread :** new, runnable(start()),running(run()), waiting(block), dead(stop())
* **wait()** will make the thread to stop and to resume use **notify();**
* using notify() it will go to **runnable state** not running state.

**April 15, 2024**

* **Collection API** -> concept  
  **Collection** -> Interface  
  **Collections** -> Class with multiple methods – different type of data structures.
* **Collection** is an **interface** interface belongs to java.util
* we can directly print collection eg: [2,4,5,6]
* we don’t have index(getIndex()) in collection api.
* Collection api works only with objects.
* To work with **index prefer using list.** If the **element is not found** in the list then it will **prints -1**.
* To support unique values use Set, if you add duplicates to it and print also it won’t print duplicates, **printing values also does not follow any sequence**, doesn’t have index support.
* Sets will not basically maintain the sequence of insertion of elements.
* If you store values in tree set then by default it will print in sorted manner ascending order. So for **sorted values in set use Treeset**.
* Iterator usage seen demo in set.
* Collection extends **iterator**.

nums is an Hashset.  
eg: Iterator<Integer> values = nums.iterator();  
while(values.hasNext())  
 System.out.println(values.next());

* **HashTable** work in the same manner as **hashMap**.
* **HashTable by default supports synchronized but hashMap doesn’t**.  
  
* **Synchronized** – we have seen in multithreading concept for avoiding race condition.  
  HashMap is non-synchronized, making it faster for single-threaded tasks, while HashTable is inherently synchronized, providing thread safety
* **Fail Fast and Fail Safe**. These iterators are very useful in exception handling.  
  The Fail fast iterator aborts the operation as soon it exposes failures and stops the entire operation. Comparatively, Fail Safe iterator doesn't abort the operation in case of a failure. Instead, it tries to avoid failures as much as possible.

**April 18, 2024**

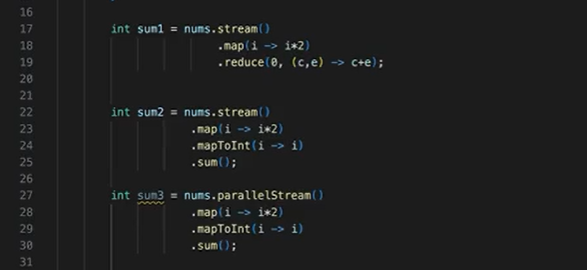
* In collections sorting can be done using Collections.sort(nums);
* If you want to sort according to your logic use comparator.
* In **comparator** interface you need to override compare method with your own logic.
* You can also sort objects using comparator.
* For natural sorting use : implements comparable<className>.
* Then you need to override compareTo method.
* If you want to give **comparison power to class** itself then use **comparable**.
* Do one handson for better understanding – done and moved to github.

**April 20, 2024**

* **Stream**:  
  not a datastructure but works with collection.  
  does not change the **original datastructure data**.

For loop, for each loop, **for each method** – works with collection only.  
**eg of for each method** :  
List<Integer> numbers = Arrays.*asList*(34,45,31,29,24);  
numbers.forEach(n -> System.*out*.println(n));

* **You can use stream only once**, if you try to use it twice then it will throw error.
* **Predicate** is a functional interface that can be used anywhere you need to evaluate a **boolean** condition.
* **Parallel stream** in java is used if you want to use streams using **multiple threads**.

  
the above three will do the same thing but in different ways.

**Parallel stream works faster than normal stream.**

* **Optional** class -> variable can be empty or it can has some values. **Optional is created to avoid null pointer exception.  
  check demo in github of yours.**
* **Method reference** : optimizes the way we use.

Eg: instead of having names.forEach(I -> System.out.println(I));   
we can use names.forEach(System.out::println);

* **Constructor reference:**   
  eg: students = names.stream().map(name -> new Student(name)).toList(); converting this to constructor reference will look like this  
  students = names.stream().map(Student::new).toList();

**Points to ponder from MCQ’s**

1. Is **method shadowing or method hiding** possible with the private method? – YES  
   The concept of method shadowing is the same as the concept of method overriding. The instance method is defined in both the classes i.e. superclass and in the subclass. When we invoke the method of the subclass, the superclass method is overridden by the subclass. The concept is known as method shadowing.

No, we cannot override private methods

1. What interface is implemented by the String, StringBuffer, and StringBuilder classes? – **CharSequence**
2. Which constructor of StringBuffer reserves **room for 16 characters** without reallocation? – **StringBuffer()**
3. Determine the capacity of the StringBuffer before and after the use of trimToSize() method in the given code:  
   StringBuffer sb = new StringBuffer("Java Code");  
   System.out.println(sb.capacity());  
   sb.trimToSize();  
   System.out.println(sb.capacity());  
     
   output : 25 9
4. How do you access the method "m1()" of the inner class inside the main method?  
   class Outer {  
   class Inner{  
   public void m1(){  
   System.out.println("inner class instance m1()");}  
   }  
   public static void main(String[] args) { }  
   }

Ans : **Outer.Inner I** = **new Outer().new Inner();**

6. Can an abstract class be instantiated directly with the new operator in Java? - NO

New points learned:

Java 8 version is also called Java 1.8 from java 9 onwards it is java 9,10,..22. Before it it is java 1.8,1.7 and all.  
Seen centralized way of handling exception in spring boot application using **@RestControllerAdvice and @ExceptionHandler.**

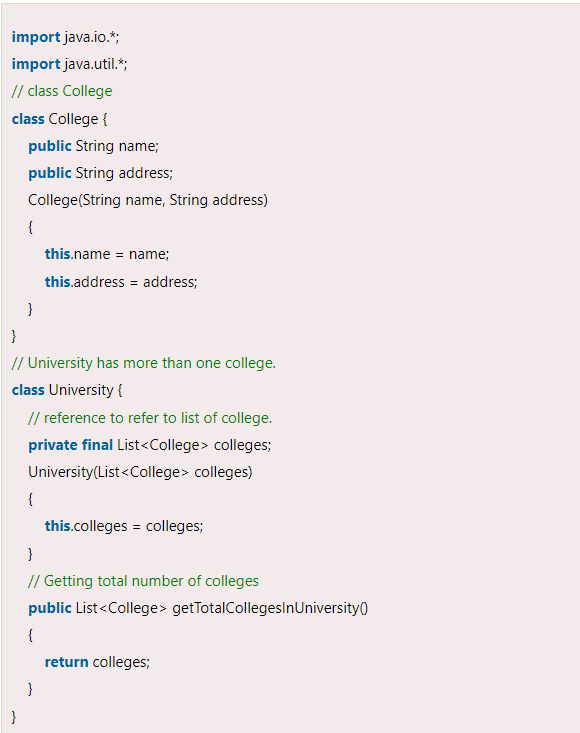
Advanced Java 1st revision on : April 21

2nd revision on : April 28

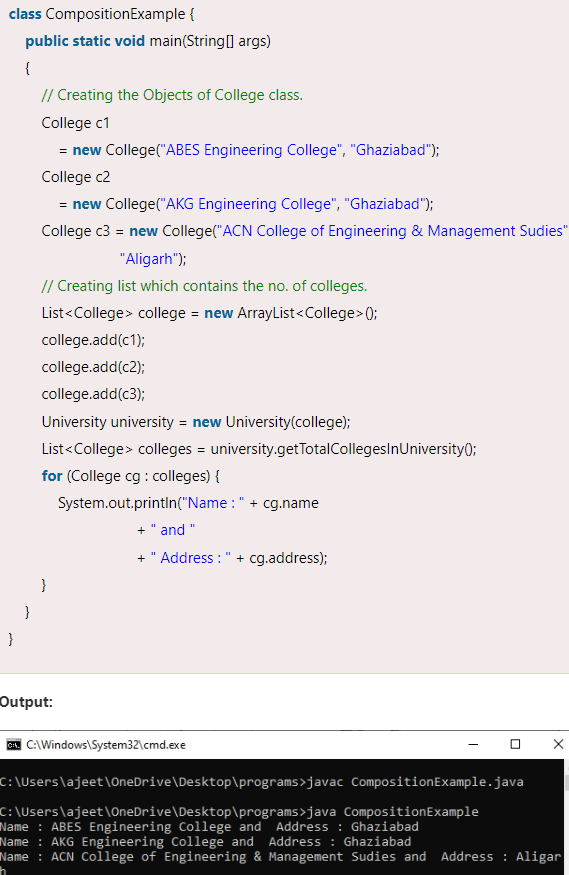
3rd revisison on June 26

May 13,2024

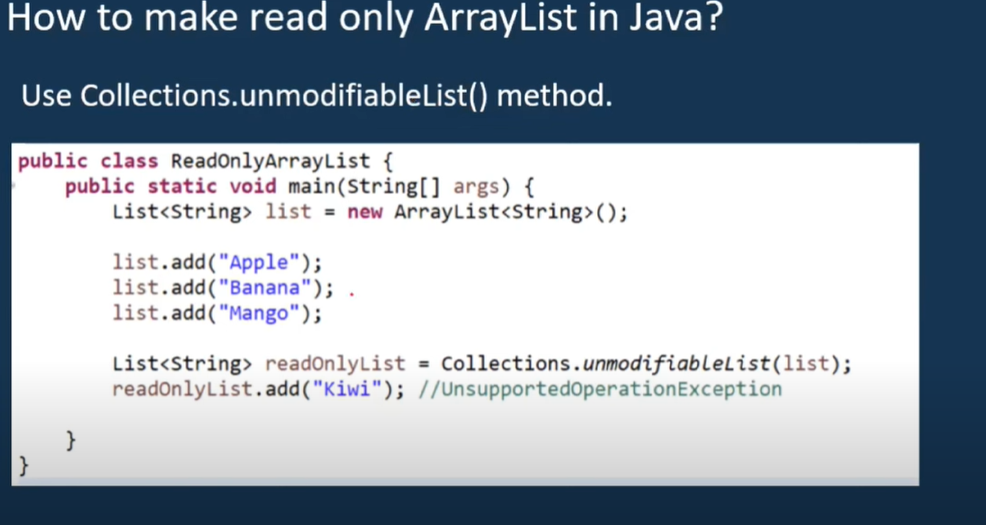
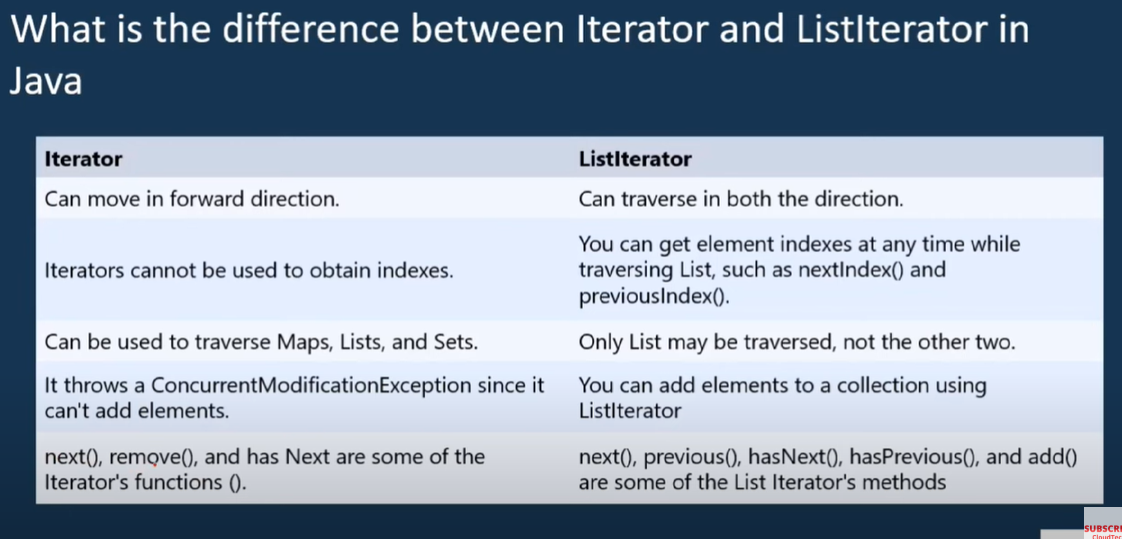
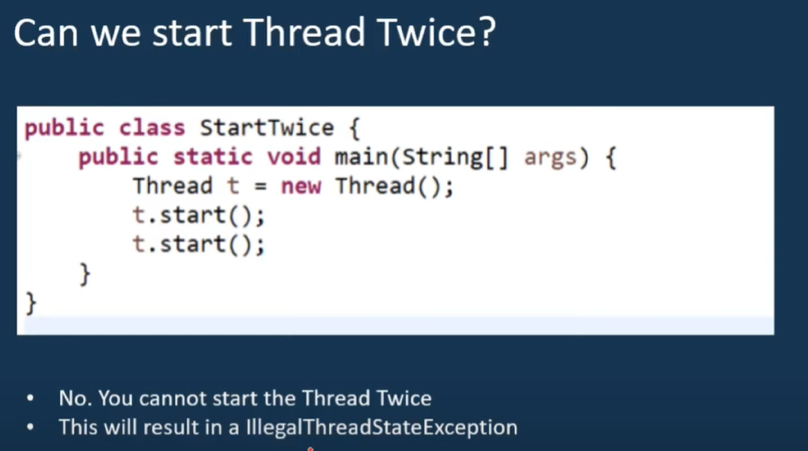
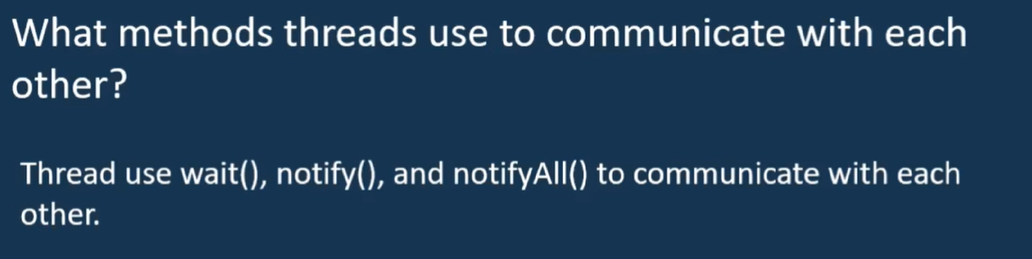
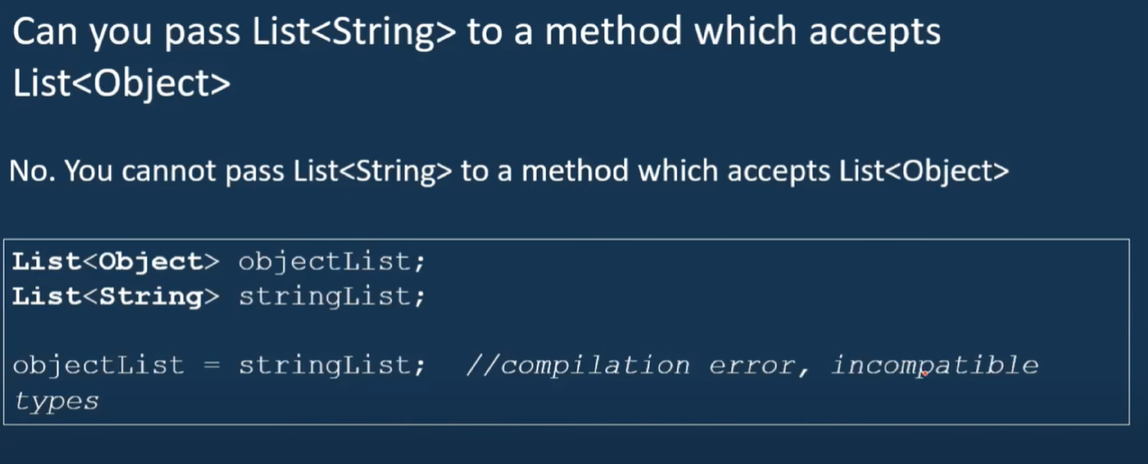
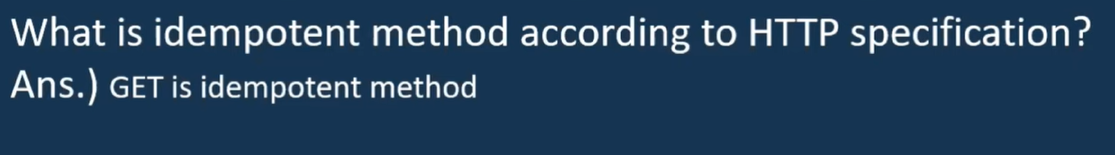
**Java interview learnings**

1. **Inheritance** provides us with **is-a** relationshipwhereas **composition** provides us with **has-a** relationship.  
   Got confused what is this is-a, has-a and what the hell is this composition that I never came across ? let us see it in explanation.  
     
   In Java, we have two types of relationship: **Is-A relationship**: Whenever one class inherits another class, it is called an IS-A relationship. **Has-A relationship**: Whenever an instance of one class is used in another class, it is called HAS-A relationship.  
     
   **Composition example :**  
   For example, if a university HAS-A college-lists, then a college is a whole, and college-lists are parts of that university. If a university is deleted then corresponding colleges for that university is also deleted.

Class contains objects of other class as it members in composition.



| **S.NO** | **Inheritance** | **Composition** |
| --- | --- | --- |
| 1. | In inheritance, we define the class which we are inheriting(super class) and most importantly it cannot be changed at runtime | Whereas in composition we only define a type which we want to use and which can hold its different implementation also it can change at runtime. Hence, Composition is much more flexible than Inheritance. |
| 2. | Here we can only extend one class, in other words more than one class can’t be extended as java do not support multiple inheritance. | Whereas composition allows to use functionality from different class. |
| 3. | In inheritance we need parent class in order to test child class. | Composition allows to test the implementation of the classes we are using independent of parent or child class. |
| 4. | Inheritance cannot extend final class. | Whereas composition allows code reuse even from final classes. |
| 5. | It is an **is-a** relationship. | While it is a **has-a** relationship. |

1. **Exchanger** is the most interesting synchronization class of Java. It facilitates the exchange of elements between a pair of threads by creating a synchronization point. It simplifies the exchange of data between two threads. Its operation is simple: it simply waits until two separate threads call its exchange() method. When that occurs, it exchanges the data supplied by the threads. It can also be viewed as a bidirectional SynchronousQueue.
2. **Features of java8**  
   Lambda expression, stream API, default methods, optional class, method references, functional interface, base64 encode decode, static methods in interface.
3. 
4. 
5. 
6. 
7. 

Idempotency in REST API means that if you send the same request again and again the result will not change after the first successful request.

Idempotent methods:

GET, HEAD, PUT, DELETE, OPTIONS, TRACE

Non-idempotent methods:

POST, PATCH

PUT is a method of modifying resource where the client sends data that updates the entire resource . PATCH is a method of modifying resources where the client sends partial data that is to be updated without modifying the entire data.

How do you start a thread until one thread finishes its execution ?

Using join()

Accenture Interview :

How do you design a singleton class ?

<https://www.geeksforgeeks.org/singleton-class-java/>

// Java program implementing Singleton class

// with using getInstance() method

// Class 1

// Helper class

class Singleton {

// Static variable reference of single\_instance

// of type Singleton

private static Singleton single\_instance = null;

// Declaring a variable of type String

public String s;

// Constructor

// Here we will be creating private constructor

// restricted to this class itself

private Singleton()

{

s = "Hello I am a string part of Singleton class";

}

// Static method

// Static method to create instance of Singleton class

public static synchronized Singleton getInstance()

{

if (single\_instance == null)

single\_instance = new Singleton();

return single\_instance;

}

}

// Class 2

// Main class

class GFG {

// Main driver method

public static void main(String args[])

{

// Instantiating Singleton class with variable x

Singleton x = Singleton.getInstance();

// Instantiating Singleton class with variable y

Singleton y = Singleton.getInstance();

// Instantiating Singleton class with variable z

Singleton z = Singleton.getInstance();

// Printing the hash code for above variable as

// declared

System.out.println("Hashcode of x is "

+ x.hashCode());

System.out.println("Hashcode of y is "

+ y.hashCode());

System.out.println("Hashcode of z is "

+ z.hashCode());

// Condition check

if (x == y && y == z) {

// Print statement

System.out.println(

"Three objects point to the same memory location on the heap i.e, to the same object");

}

else {

// Print statement

System.out.println(

"Three objects DO NOT point to the same memory location on the heap");

}

}

}