1. what is functional interface.?

* interface that contains only one abstract method. They can have only one functionality to exhibit.
* Can have any number of default and static method.
* Lambda expression can be used to represent the instance of a functional interface.

1. g:-public interface Drawable{

Public static void draw1(){}

Public default void draw2(){}

Public void draw3();

}

1. why default and static methods are added in java8 in Interface.?

* Prior to java 8, interface in java can only have abstract methods. ... Java 8 allows the interfaces to have default and static methods.
* The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

E.g:because the Java 8 API introduces many new methods on existing interfaces, such as the sort method on the List interface

default void sort(Comparator<? super E> c){

Collections.sort(this, c);

}

1. Abstract class Vs Functional Interface

* Abstract classes have no restrictions on field and method modifiers,while in an interface, all are public by default.
* We can have instance and static initialization blocks in an abstract class, whereas we can never have them in the interface.Abstract classes may also have constructors which will get executed during the child object's instantiation.
* Functional Interface Any interface with a single abstract method other than static and default methods is considered a functional interface. We can use this feature to restrict the number of abstract methods to be declared. While in abstract classes, we can never have this restriction on the number of abstract methods declaration.Default methods only allowed in interface.

E.g:Public abstract class Test{

**int** i;

**static** **int** *j*;

**static** {}

**public** Test() {}

**public** **static** **void** draw1(){}

**public** **abstract** **void** draw3();

}

Public interface class Test{

**public** **static** **final** **int** ***i*** = 0;

**public** **static** **void** draw1(){}

**public** **default** **void** draw2(){}

**public** **abstract** **void** draw10();

}

1. what is Lambda expression .

* Lambda expressions basically express instances of [functional interfaces](https://www.geeksforgeeks.org/functional-interfaces-java/) (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces.
* Enable to treat functionality as a method argument, or code as data.
* A function that can be created without belonging to any class.
* A lambda expression can be passed around as if it was an object and executed on demand.

E.g:

**interface** FuncInterface

{ // An abstract function

**void** abstractFun(**int** x);

// A non-abstract (or default) function

**default** **void** normalFun()

{

System.***out***.println("Hello");

}}

**class** Test{

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

{

FuncInterface fobj = (**int** x)->System.***out***.println(2\*x);

fobj.abstractFun(5);

}}

Class test2{

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

ArrayList<Integer> arrL = **new** ArrayList<Integer>();

arrL.add(1);

arrL.add(2);

arrL.add(3);

arrL.add(4);

arrL.forEach(n -> System.out.println(n));

arrL.forEach(n -> { **if** (n%2 == 0) System.out.println(n); });

}

}

1. what is streamApi

* A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
* To Process element present inside collection.bulk operations.
* Stream is the default function present Collections interface.
* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.
* Stream is a Interface.

E.g

public interface **Stream<T>**

Stream s = Collections.stream();

Different Operations On Streams-  
**Intermediate Operations:**

1. **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.

E.g:  
List<Integer> number = Arrays.*asList*(2,3,4,5);

List square = number.stream().map(x->x\*x).collect(Collectors.*toList*());

square.forEach(x->System.***out***.println(x));

1. **filter:** The filter method is used to select elements as per the Predicate passed as argument.

E.g   
List <String> names = Arrays.*asList*("Reflection","Collection","Stream");

List<String> result = names.stream().filter(r->r.startsWith("S"))

.collect(Collectors.*toList*());

result.forEach(x->System.***out***.println(x));

1. **sorted:** The sorted method is used to sort the stream.

E.g   
List names2 = Arrays.*asList*("Reflection","Collection","Stream");

List result3 = names.stream().sorted().collect(Collectors.*toList*());

result.forEach(x->System.***out***.println(x));

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
   E.g:

List <Integer> number2 = Arrays.*asList*(2,3,4,5,3);

Set square1 = number2.stream().map(x->x\*x).collect(Collectors.*toSet*());

square1.forEach(x->System.***out***.println(x));

1. **forEach:** The forEach method is used to iterate through every element of the stream.

E.g:

square1.forEach(x->System.***out***.println(x));

1. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
   The reduce method takes a BinaryOperator as a parameter.

E.g:

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

1. **Why streams Api are Lazy?.**

* Streams are lazy because intermediate operations are not evaluated unless terminal operation is invoked.
* Each intermediate operation creates a new stream, stores the provided operation/function and return the new stream. The pipeline accumulates these newly created streams.
* The time when terminal operation is called, traversal of streams begins and the associated function is performed one by one.

1. **Difference between Map() and flatMap()**

* Stream interface hasa[map()](https://www.geeksforgeeks.org/stream-map-java-examples/) and [flatmap()](https://www.geeksforgeeks.org/stream-flatmap-java-examples/) methods and both have intermediate stream operation and return another stream as method output.
* Both of the functions map() and flatMap are used for transformation and mapping operations.
* **map()**can be used where we have to map the elements of a particular collection to a certain function, and then we need to return the stream which contains the updated results.
* **flatMap()** can be used where we have to flatten or transform out the string, as we cannot flatten our string using map().
* **Example:** Getting the 1st Character of all the String present in a List of Strings and returning the result in form of a stream.

E.g: A) Map().

ArrayList<String> fruit = **new** ArrayList<>();

fruit.add("Apple");fruit.add("mango");fruit.add("pineapple");

fruit.add("kiwi");

List list = fruit.stream().map(s -> s.length()).collect(Collectors.toList());

System.out.println(list);

OutPut:-[5, 5, 9, 4]

E.g: B) flatMap().

List<List<Integer> > number = **new** ArrayList<>();

number.add(Arrays.asList(1, 2));

number.add(Arrays.asList(3, 4));

number.add(Arrays.asList(5, 6));

number.add(Arrays.asList(7, 8));

System.out.println(“before flatMap method call = ”+number);

List<Integer> flatList= number.stream()

.flatMap(list -> list.stream()).collect(Collectors.toList());

System.out.println(“After flatMap method call = ”+flatList);

outPut:-before flatMap method call = [[1, 2], [3, 4], [5, 6], [7, 8]]

After flatMap method call = [1, 2, 3, 4, 5, 6, 7, 8]

1. ArrayList vs LinkedList

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a ****dynamic array**** to store the elements. | LinkedList internally uses a ****doubly linked list**** to store the elements. |
| 2) Manipulation with ArrayList is ****slow**** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is ****faster**** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory. |
| 3) An ArrayList class can ****act as a list**** only because it implements List only. | LinkedList class can ****act as a list and queue**** both because it implements List and Deque interfaces. |
| 4) ArrayList is ****better for storing and accessing**** data. | LinkedList is ****better for manipulating**** data. |

1. Advantage of LinkedList over ArrayList.?

* ArrayList provides constant time for search operation so it is better to use ArrayList if searching is frequent operation.Than add of remove operation.
* LinkedList provide constant time for add and Remove operation so it is better to use linkedList for manipulation
* LinkedList class also implement deque interface so you can get the functionality of double-ended queue in the LinkedList.ArraList Class doesn’t implement deque interface.
* In Case of ready-only collections or collections which are rarely modified ArrayList is suitable.

10) Hashmap vs linkedHashmap or How you can maintain insertion order for hashMap.

* The **LinkedHashMap** is just like [HashMap](https://www.geeksforgeeks.org/java-util-hashmap-in-java/) with an additional feature of maintaining an order of elements inserted into it.
* HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

**Important Features of a LinkedHashMap are listed as follows:**

* A LinkedHashMap contains values based on the key. It implements the Map interface and extends the HashMap class.
* It contains only unique elements.
* It may have one null key and multiple null values.
* It is non-synchronized.
* It is the same as HashMap with an additional feature that it maintains insertion order. For example, when we run the code with a HashMap, we get a different order of elements.
* The implementation of the LinkedHashMap is very similar to a [doubly-linked list](https://www.geeksforgeeks.org/doubly-linked-list/).

E.g :

Linked HashMap Node Structure HashMap Node Structure

Class Node{ class{

int hash; int hash;

K key; K key;

V value; V value;

Node next; Node next;

Node previous; }

}

HashMap<String,Integer> marks= **new** HashMap<>();

LinkedHashMap<String,Integer> lhm=**new** LinkedHashMap<>();

marks.put("Bob",100);marks.put("Rob", 200);marks.put("Glob", 300);

marks.put("Slob", 400);marks.put("Ylob", 600);marks.put("Sob", 700);

lhm.put("Bob",100);lhm.put("Rob", 200);lhm.put("Glob", 300);

lhm.put("Slob", 400);lhm.put("Ylob", 600);lhm.put("Sob", 700);

System.***out***.println("marks HashMap"+marks);

System.***out***.println("lhm LinkedHashMap"+lhm);

OutPut:-marks

HashMap{Bob=100, Rob=200, Sob=700, Glob=300, Slob=400, Ylob=600}

lhm LinkedHashMap{Bob=100, Rob=200, Glob=300, Slob=400, Ylob=600, Sob=700}

11)Difference between hashMap and concurrent hashMap?what is ConcurrentModificationException.?

* HashMap is non-Synchronized in nature i.e. HashMap is not Thread-safe whereas ConcurrentHashMap is Thread-safe in nature.
* HashMap performance is relatively high because it is non-synchronized in nature and any number of threads can perform simultaneously. But ConcurrentHashMap performance is low sometimes because sometimes Threads are required to wait on ConcurrentHashMap.
* While one thread is Iterating the HashMap object, if other thread try to add/modify the contents of Object then we will get Run-time exception saying **ConcurrentModificationException**.Whereas In ConcurrentHashMap we wont get any exception while performing any modification at the time of Iteration.

12)why to use concurrent HashMap when their is a synchronized Map ?

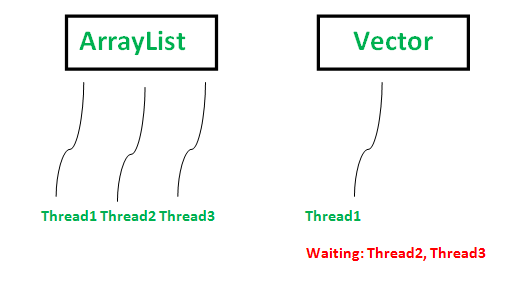
* The major advantage in case of SynchronizedMap every write operation acquires Lock on entire map Object.While in case of ConcurrentHashMap

The lock is acquire only to one segment/bucket/index.

* ConcurrentHashMap doesn’t allow inserting null as a key or value.Synchronized HashMap allows inserting null as a key.
* ConccurentHashMap doesn’t throw Concurrent Modification Exception.Synchronized HashMap throw ConcurrentModificationException.

1. Diffrence Between vector and arrayList?

| ArrayList | Vector |
| --- | --- |
| ArrayList increments 50% of the current array size if the number of elements exceeds ts capacity. | Vector increments 100% means doubles the array size if the total number of elements exceeds its capacity. |
| ArrayList is fast because it is non-synchronized. | Vector is slow because it is synchronized, i.e., in a multithreading environment, it holds the other threads in a runnable or non-runnable state until the current thread releases the lock of the object. |



1. Array vs arraylist?

* Array is a fixed length data structure whereas ArrayList is a variable length Collection class. We cannot change length of array once created in Java but ArrayList can be changed.
* We cannot store primitives in ArrayList, it can only store objects. But array can contain both primitives and objects in Java. Since Java 5, primitives are automatically converted in objects which is known as auto-boxing.

1. Can we store null in treeset.?

* No.TreeSet does not allows to store any null in java. Any attempt to add null throws runtimeException (NullPointerException). For storing elements internally uses a self-balancing tree.

1. What is Iterator? how and why to use it.

* A Java Cursor is an Iterator, which is used to iterate or traverse or retrieve a Collection or Stream object’s elements one by one. There are **three cursors in Java**.
* The primary purpose of an iterator is to allow a user to process every element of a container while isolating the user from the internal structure of the container. This allows the container to store elements in any manner it wishes while allowing the user to treat it as if it were a simple sequence or list.

1. Iterator
2. Enumeration
3. ListIterator

***Note:****SplitIterator can also be considered as a cursor as it is a type of Iterator only.it is implemented inside Iterator* Spliterator acts as the intermediate while creating Sequential Stream from Iterator.

***A)Iterator***

* Iterators in Java are used in the [Collection framework](https://www.geeksforgeeks.org/collections-in-java-2/) to retrieve elements one by one. It is a **universal** iterator as we can apply it to any Collection object.
* By using Iterator, we can perform both read and remove operations. It is an improved version of Enumeration with the additional functionality of removing an element.
* Iterator is the **only** cursor available for the entire collection framework.E.g Iterator itr = collectionObject.**iterator**();

1. Enumeration

* It is used to iterate only Legacy Collection classes.
* It supports only READ operation.
* Iterator and Enumeration are Uni-directional Java Cursors which means support only Forward Direction Iteration.

1. **ListIterator**

* It is only applicable for List collection implemented classes like ArrayList, LinkedList, etc.
* It provides bi-directional iteration.
* ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

1. g ListIterator ltr = listObject.**listIterator**();
2. Fail fast and fail safe Iterator.?

**Fail fast**

* The Fail Fast system is a system that shuts down immediately after an error is reported. All the operations will be aborted instantly in it.
* The iterator in Java is used to traverse over a collection's objects. The collections return two types of iterators, either it will be Fail Fast or Fail Safe.
* The Fail Fast iterators immediately throw ConcurrentModificationException in case of structural modification of the collection. Structural modification means adding, removing, updating the value of an element in a data collection while another thread is iterating over that collection.   
  E.g Fail Fast iterator are iterator on ArrayList, HashMap collection classes.
* **How It Work** :-The Fail Fast iterator uses an internal flag called ****modCount**** to know the status of the collection, whether the collection is structurally modified or not. The modCount flag is updated each time a collection is modified; it checks the next value; if it finds, then the modCount will be modified after this iterator has been created. It will throw ConcurrentModificationException.
* The Fail Fast iterator uses an original collection to traverse over the collection's elements.
* They are memory savers, don't require extra memory.
* The Fail Fast iterators returned by ArrayList, HashMap, Vector classes.

## **Fail Safe Iterator**

* A fail-safe iterator does not throw any exceptions unless it can handle if the collection is modified during the iteration process. This can be done because they operate on the copy of the collection object instead of the original object. The structural changes performed on the original collection ignored by them and affect the copied collection, not the original collection. So, the original collection will be kept structurally unchanged.
* It is not necessary that a collection that does not use the Fail Fast iterator create a clone or copy of it in memory to avoid the[ConcurrentModificationException](https://www.javatpoint.com/concurrentmodificationexception-in-java).For example, the [ConcurrentHashMap](https://www.javatpoint.com/java-concurrenthashmap) does not operate on a separate copy of an object, although it does not fail fast. Instead, it uses semantics that is specified by the specification as non-fail fast iteration.
* We can perform the modification operations on a collection while iterating over it.
* They will not throw ConcurrentModificationException during the iteration.
* The Fail Safe iterators use a copy of the collection to traverse over the elements.
* Unlike the Fail Fast, they require more memory as they cloned the collection.
* The examples of Fail Safe iterators are ConcurrentHashMap, CopyOnWriteArrayList, etc.

**E.g 1)** **Where copy of object is created:**

 CopyOnWriteArrayList<Integer> list

            = **new** CopyOnWriteArrayList<Integer>(**new** Integer[] { 1, 7, 9, 11 });

        Iterator itr = list.iterator();

**while** (itr.hasNext()) {

            Integer i = (Integer)itr.next();

            System.out.print(i);

**if** (i == 7)

 list.add(15); // It will not be printed  //This means it has created a separate copy of the collection

        }

OutPut :-1 7 9 11

**E.g 2) Where copy of object is not created:**

 ConcurrentHashMap<String, Integer> m

            = **new** ConcurrentHashMap<String, Integer>();

        m.put("ONE", 1); //Adding values   m.put("SEVEN", 7);   m.put("FIVE", 5);   m.put("EIGHT", 8);

       Iterator it = m.keySet().iterator();

**while** (it.hasNext()) {

            String key = (String)it.next();

            System.out.println(key + " : " + m.get(key));    // This will reflect in iterator.   // This means it has not created separate copy of objects

m.put("NINE", 9);}

OutPut:- EIGHT : 8

FIVE : 5

NINE : 9

ONE : 1

SEVEN : 7

### 18) What is the difference between Collection and Collections?

* The Collection is an interface whereas Collections is a class.
* The Collection interface provides the standard functionality of data structure to List, Set, and Queue. However, Collections class is to sort and synchronize the collection elements.
* The Collection interface provides the methods that can be used for data structure whereas Collections class provides the static methods which can be used for various operation on a collection.

19)String Literal vs new

**String literal**

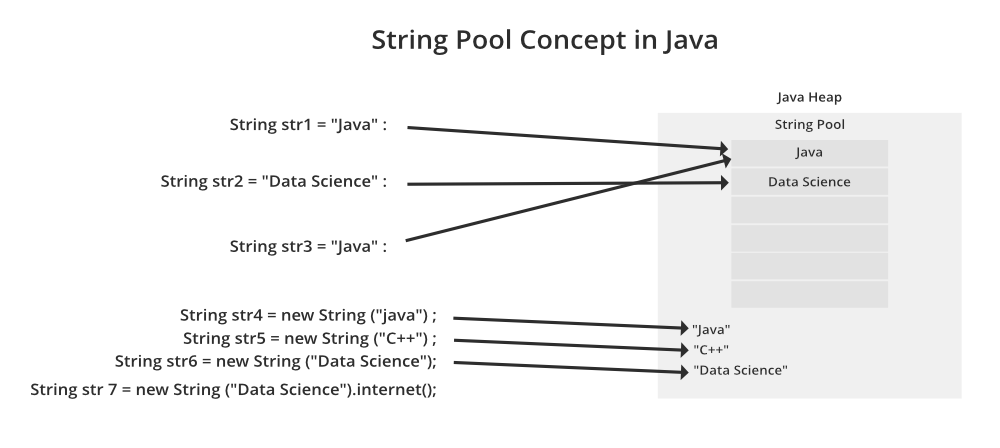
*String str = “GeeksForGeeks”;*

* This is string literal(String s=”some”). When you declare string like this, you are actually calling intern() method on String. This method **references internal pool** of string objects. If there already exists a string value “GeeksForGeeks”, then str will reference of that string and no new String object will be created.

**String Object**

*String str = new String(“GeeksForGeeks”);*

* This is string object. In this method JVM is forced to create a new string reference, even if “GeeksForGeeks” is in the reference pool.
* Therefore, if we compare performance of string literal and string object, string object will always take more time to execute than string literal because it will construct a new string every time it is executed.



1. What is Association,Aggregation & composition.?

* Association is a relation between two separate classes which establishes through their Objects.
* **Composition** and **Aggregation** are the two forms of association.
* Aggregation :- represents Has-A’s relationship.
* It is a **unidirectional association** i.e. a one-way relationship. For example, a 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 affect the other entity.

***When do we use Aggregation ??****Code reuse is best achieved by aggregation.*

* 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 entities are dependent on each other.
* When there is a composition between two entities, the composed object **cannot exist** without the other entity.

1. What is comparable and comparator.?

* Java Comparable interface is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object obj).
* you can sort the elements on the basis of single data member only.For example, it may be rollno, name, age or anything else.
* String class and Wrapper classes implement the Comparable interface by default. So if you store the objects of string or wrapper classes in a list, set or map, it will be Comparable by default.

E.g  **public** **int** compareTo(Student st){

**if**(age==st.age)

**return** 0;

**else** **if**(age<st.age)

**return** 1;

**else**

**return** -1;  }

* Comparator provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Example:-**class** AgeComparator **implements** Comparator<Student>{

**public** **int** compare(Student s1,Student s2){

**if**(s1.age==s2.age)

**return** 0;

**else** **if**(s1.age>s2.age)

**return** 1;

**else**

**return** -1;

}

}

**class** NameComparator **implements** Comparator<Student>{

**public** **int** compare(Student s1,Student s2){

**return** s1.name.compareTo(s2.name);

}

}

1. serialization /deserialization

* Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.
* The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.
* To make a Java object serializable we implement the **java.io.Serializable** interface.  
  The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

public final void writeObject(Object obj)

throws IOException

The ObjectInputStream class contains **readObject()** method for deserializing an object.

public final Object readObject()

throws IOException,

ClassNotFoundException

**Points to remember**  
1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.  
2. Only non-static data members are saved via Serialization process.  
3. Static data members and transient data members are not saved via Serialization process.So, if you don’t want to save value of a non-static data member then make it transient.  
4. Constructor of object is never called when an object is deserialized.  
5. Associated objects must be implementing Serializable interface.

* The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and receiver of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the receiver has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name.
* It must be static, final and of type long.  
  i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;
* The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.

Can we catch multiple exceptions in single catch block

1. what is Split Iterator.?

* Spliterators, like other Iterators, are for traversing the elements of a source. A source can be a [Collection](https://www.geeksforgeeks.org/collections-in-java-2/), an [IO channel](https://docs.oracle.com/javase/7/docs/api/java/nio/channels/Channels.html) or a [generator function](https://en.wikipedia.org/wiki/Generator_(computer_programming)).
* It is included in JDK 8 for support of efficient parallel. traversal(parallel programming) in addition to sequential traversal.
* However, you can use Spliterator even if you won’t be using parallel execution. One reason you might want to do so is because it combines the hasNext and next operations into one method.

E.g

List<String> list = **new** ArrayList<String>();

list.add("ram");

list.add("krishna");

list.add("shankar");

list.add("karan");

list.add("arjun");

list.add("bunty");

list.add("ramesh");

list.add("suresh");

Spliterator<String> trysplit1=list.spliterator();

Spliterator<String> trysplit2=trysplit1.trySplit();

System.***out***.println("OutPut of try split1");

trysplit1.forEachRemaining(s->System.***out***.println(s));

System.***out***.println("OutputFrom of try split2");

trysplit2.forEachRemaining(s->System.***out***.println(s));

outPut:-

OutPut of try split1

arjun

bunty

ramesh

suresh

OutputFrom of try split2

ram

krishna

shankar

Karan

24)Difference between Serialization and Sychronization.?

* Synchronization refers to multi-threading. A synchronized block of code can only be executed by one thread at a time.
* Serialization refers to converting objects to byte streams either for storage or transmission. The act of serialization encodes the data according to specific rules.

1. Difference between yield() & Join() method.?

A)Yield()

Suppose there are three threads t1, t2, and t3. Thread t1 gets the processor and starts its execution and thread t2 and t3 are in Ready/Runnable state. The completion time for thread t1 is 5 hours and the completion time for t2 is 5 minutes. Since t1 will complete its execution after 5 hours, t2 has to wait for 5 hours to just finish 5 minutes job. In such scenarios where one thread is taking too much time to complete its execution, we need a way to prevent the execution of a thread in between if something important is pending. yield() helps us in doing so.

The **yield()**basically means that the thread is not doing anything particularly important and if any other threads or processes need to be run, they should run. Otherwise, the current thread will continue to run.

**Use of yield method:**

* Whenever a thread calls**java.lang.Thread.yield** method gives hint to the thread scheduler that it is ready to pause its execution. The thread scheduler is free to ignore this hint.
* If any thread executes the yield method, the thread scheduler checks if there is any thread with the same or high priority as this thread. If the processor finds any thread with higher or same priority then it will move the current thread to Ready/Runnable state and give the processor to another thread and if not – the current thread will keep executing.
* Once a thread has executed the yield method and there are many threads with the same priority is waiting for the processor, then we can’t specify which thread will get the execution chance first.
* The thread which executes the yield method will enter in the Runnable state from Running state.
* Once a thread pauses its execution, we can’t specify when it will get a chance again it depends on the thread scheduler.

1. Join()

* **java.lang.Thread** class provides the join() method which allows one thread to wait until another thread completes its execution. If **t** is a Thread object whose thread is currently executing, then **t.join()** will make sure that **t** is terminated before the next instruction is executed by the program.  
  If there are multiple threads calling the join() methods that means overloading on join allows the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

|  |
| --- |
| **class** ThreadJoining **extends** Thread  {      @Override  **public** **void** run()      {  **for** (**int** i = 0; i < 2; i++)          {  **try**              {                  Thread.sleep(500);                  System.out.println("Current Thread: "                          + Thread.currentThread().getName());              }    **catch**(Exception ex)              {                  System.out.println("Exception has" +                                  " been caught" + ex);              }              System.out.println(i);          }      }  }    **class** GFG  {  **public** **static** **void** main (String[] args)      {            // creating two threads          ThreadJoining t1 = **new** ThreadJoining();          ThreadJoining t2 = **new** ThreadJoining();          ThreadJoining t3 = **new** ThreadJoining();            // thread t1 starts          t1.start();            // starts second thread after when          // first thread t1 has died.  **try**          {              System.out.println("Current Thread: "                    + Thread.currentThread().getName());              t1.join();          }    **catch**(Exception ex)          {              System.out.println("Exception has " +                                  "been caught" + ex);          }            // t2 starts          t2.start();            // starts t3 after when thread t2 has died.  **try**          {              System.out.println("Current Thread: "                   + Thread.currentThread().getName());              t2.join();          }    **catch**(Exception ex)          {              System.out.println("Exception has been" +                                      " caught" + ex);          }            t3.start();      }  } |

output:

Current Thread: main

Current Thread: Thread-0

0

Current Thread: Thread-0

1

Current Thread: main

Current Thread: Thread-1

0

Current Thread: Thread-1

1

Current Thread: Thread-2

0

Current Thread: Thread-2

1

1. what is daemon thread and how to create.?

Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

**Example of Daemon Thread in Java:**Garbage collection in Java (gc), finalizer, etc.

**Properties of Java Daemon Thread**

* They can not prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution.
* If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. JVM does not care whether the Daemon thread is running or not.
* It is an utmost low priority thread.
* *Whenever the last non-daemon thread terminates, all the daemon threads will be terminated automatically.*

*E.g*

**public** **class** DaemonThread **extends** Thread

{

**public** DaemonThread(String name){

**super**(name);

    }

**public** **void** run()

    {

        // Checking whether the thread is Daemon or not

**if**(Thread.currentThread().isDaemon())

        {

            System.out.println(getName() + " is Daemon thread");

        }

**else**

        {

            System.out.println(getName() + " is User thread");

        }

    }

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

    {

        DaemonThread t1 = **new** DaemonThread("t1");

        DaemonThread t2 = **new** DaemonThread("t2");

        DaemonThread t3 = **new** DaemonThread("t3");

        // Setting user thread t1 to Daemon

        t1.setDaemon(**true**);

        // starting first 2 threads

        t1.start();

        t2.start();

        // Setting user thread t3 to Daemon

        t3.setDaemon(**true**);

        t3.start();

    }

}

**Output:**

t1 is Daemon thread

t3 is Daemon thread

t2 is User thread

1. Exception Checked and unchecked ?



## Checked Exceptions

* Theseare the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the *[throws](https://www.geeksforgeeks.org/throw-throws-java/)*[keyword](https://www.geeksforgeeks.org/throw-throws-java/).
* In general, checked exceptions represent errors outside the control of the program. For example, the constructor of [FileInputStream](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/FileInputStream.html" \l "<init>(java.io.File)) throws *FileNotFoundException*if the input file does not exist.

For example, consider the following Java program that opens the file at location “C:\test\a.txt” and prints the first three lines of it. The program doesn’t compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException.*

**class** GFG {

    // Main driver method

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

    {

        // Reading file from path in local directory

        FileReader file = **new** FileReader("C:\\test\\a.txt");

        // Creating object as one of ways of taking input

        BufferedReader fileInput = **new** BufferedReader(file);

        // Printing first 3 lines of file "C:\test\a.txt"

**for** (**int** counter = 0; counter < 3; counter++)

            System.out.println(fileInput.readLine());

        // Closing file connections

        // using close() method

        fileInput.close();

    }

}

Output:-

*Exception in thread "main" java.lang.Error: Unresolved compilation problems:*

*Unhandled exception type FileNotFoundException*

*Unhandled exception type IOException*

*Unhandled exception type IOException*

*at Exception.main(Exception.java:10)*

To fix the above program, we either need to specify a list of exceptions using throws, or we need to use a try-catch block. We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

Some [common checked exceptions](https://www.baeldung.com/java-common-exceptions) in Java are IOException, SQLException and ParseException.

The [Exception](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Exception.html) class is the superclass of checked exceptions, so we can [create a custom checked exception](https://www.baeldung.com/java-new-custom-exception) by extending Exception:

Public **class** IncorrectFileNameException **extends** Exception {

**public** IncorrectFileNameException(String errorMessage) {

**super**(errorMessage);

}

}

## Unchecked Exceptions

If a program throws an unchecked exception, it reflects some error inside the program logic.

For example, if we divide a number by 0, Java will throw ArithmeticException: