Amold – Java Guide

Java

1. Wrapper classes

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper Methods
================
ValueOf => convert String into wrapper
parseXXX => convert String into premitive

Auto boxing /Unboxing
====================
Auto boxing --> Integer i=10;
Auto unboxing --> int i=new Integer (10)

2. String

String is basically an object that represents sequence of char values.

```
String s1=new String("you can not change me"); //created in heap
String s2=new String("you can not change me"); //created in heap

System.out.println(s1==s2) //false

String s3="you can not change me"; //created in SCP
String s4="you can not change me"; //refence to existing obj s3

System.out.println(s1==s3); //false

String s5="you can not"+" change me";
System.out.println(s3==s5); //true

String s6="you can not";
String s7=s6+" change me";

System.out.println(s7==s3); //false

final String s8="you can not";
String s9=s8+" change me";

System.out.println(s9==s3); //true0
```

Advantage of SCP :

1. Rather than creating separate object , create one object and assign reference to it.

2.it helps for memory utilization and performance improvement.

Disadvantage of SCP .

we have to use string immutability always.

>because of some runtime operation performed on string object it will be created in heap memory

```
String s6="you can not";
String s7=s6+" change me";
```

> final/constant + final/constant = create String in SCP

```
final String s8="you can not";
String s9=s8+" change me";
```

- Even if we break the reference of string still it persists into the SCP area due to which it becomes a security concern so to storing security information use character arrays instead of string.

Overloading Vs Overriding

Method **Overloading** Rules

Two methods will be treated as overloaded if both follow the mandatory rules below:

- Both must have the same method name.
- Both must have different argument lists.

And if both methods follow the above mandatory rules, then they may or may not:

- Have different return types.
- Have different access modifiers.
- Throw different checked or unchecked exceptions.

Method **Overriding** Rules

When a method in a subclass has the **same name**, **same parameters** or **signature**, and **same return type**(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

- **Overriding and Access-Modifiers** : The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass.
- **Final methods can not be overridden** : If we don't want a method to be overridden, we declare it as final
- **Static methods can not be overridden(Method Overriding vs Method Hiding)** : When you define a static method with same signature as a static method in base class, it is known as method hiding.
- **Private methods can not be overridden** : Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in a subclass.
- **The overriding method must have same return type (or subtype)** : From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomena is known as **covariant return type**.
- **Overriding and constructor** : We can not override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name)
- **Overriding and synchronized/strictfp method :** The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it's possible that a synchronized/strictfp method can override a non synchronized/strictfp one and vice-versa.
- **Overriding and Exception-Handling** :
  Below are two rules to note when overriding methods related to exception-handling.

Rule#1 : If the super-class overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

Rule#2 : If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

```
class Parent {
   void m1() {     System.out.println("From parent m1()");  }
   void m2(){     System.out.println("From parent  m2()"); }
}
class Child extends Parent {

   @Override  // no issue while throwing unchecked exception
   void m1() throws ArithmeticException
   {  System.out.println("From child m1()");   }

   @Override  // compile-time error - issue while throwing checked exception
   void m2() throws Exception {
          System.out.println("From child m2");    }
}
```

hashcode & equals contract

It is generally necessary to override the hashCode() method whenever equals() method is overridden,
so as to maintain the general contract for the hashCode() method,
which states that **equal objects must have equal hash codes.**

If two objects are equal according to the equals(Object) method,
then calling the hashCode method on each of the two objects must produce the same integer result.

```
@Override
public int hashCode() {
	final int prime = 31; // product of prime number with another number gives best posibilities return unique number
	int result = 1;
	result = prime * result + id;
	result = prime * result + ((name == null)? 0: name.hashCode());
	result=prime*result+ (isBoolean? 127731: 127735);
	return result;
}

// Hashcode - prime * result + data(hashcode)
```

```
@Override
public boolean equals(Object obj) {
	if (this == obj)
		return true;
	if (obj == null)
		return false;
	if (getClass() != obj.getClass())
		return false;
	Employee other = (Employee) obj;
	if (id != other.id)
		return false;
	if (name == null) {
		if (other.name != null)
			return false;
	} else if (!name.equals(other.name))
		return false;
	return true;
}
```

Immutable Class

1.Declare the class as final so it can’t be extended.
2.Make all fields private so that direct access is not allowed.
3.Don’t provide setter methods for variables.
4.Make all mutable fields final so that its value can be assigned only once.
5.Initialize all the fields via a constructor performing deep copy.
6.Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference.

final + private + no setter() + clone(constructor + getter// new Date().clone())

```
	final class Person {

		private int id;
		private String name;
		private Date date;

		public Person(int id, String name, Date date) {
			super();
			this.id = id;
			this.name = name;
			this.date = (Date) date.clone();
		}

		@Override
		public String toString() {
			return "Person [id=" + id + ", name=" + name + ", date=" + date + "]";
		}

		public int getId() {
			return id;
		}

		public String getName() {
			return name;
		}

		public Date getDate() {
			return (Date) date.clone();
		}

	}
```

Threads

Creation of threads in java

```
class Multi3 implements Runnable{
        public void run(){  //statements }
}


class Multi extends Thread{
        public void run(){  //statements }
}
```

Important Method

yeild()	wait()
sleep()	notify()
join()	notifyAll()

1. **public static native void yield()** -- > checks priority
   A **yield()** method is a **static** method of **Thread** class and it can stop the currently executing thread and will give a chance to **other waiting threads of the same priority.** If in case there are no waiting threads or if all the waiting threads have **low priority** then the same thread will continue its execution

2. **public static void sleep(long millis) throws InterruptedException**
   public static void sleep(long millis, int nanos) throws InterruptedException
   sleep thread for the specified number of milliseconds & nanoseconds.

3. **public final void join() throws InterruptedException**
   public final void join(long millis) throws InterruptedException
   it can be used to join the start of a thread's execution to the end of another thread's execution so that a thread will not start running until another thread has ended

how to execute threads sequentially

There are three threads T1, T2, and T3? How do you ensure sequence T1, T2, T3 in Java?

Sequencing in multi-threading can be achieved by different means but you can simply use the join() method of thread class to start a thread when another one has finished its execution. To ensure three threads execute you need to start the last one first e.g. T3 and then call join methods in reverse order e.g. T3 calls T2. join and T2 calls T1.join, these ways T1 will finish first and T3 will finish last.

Interupt :

Public void Interupt()

The **interrupt()** method of thread class is used to interrupt the thread. If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked) then using the interrupt() method, we can interrupt the thread execution by throwing InterruptedException.

If the thread is not in the sleeping or waiting state then calling the interrupt() method performs a normal behavior and doesn't interrupt the thread but sets the interrupt flag to true.

wait /notfy/notifyAll – Object methods

The threads can communicate with each other through wait(), notify() and notifyAll() methods in Java. These are final methods defined in the Object class and can be called only from within a synchronized context. The wait() method causes the current thread to wait until another thread invokes the notify() or notifyAll() methods for that object. The notify() method wakes up a single thread that is waiting on that object's monitor. The notifyAll() method wakes up all threads that are waiting on that object's monitor.

Thread Safety

1. Synchronization is the easiest and most widely used tool for thread safety in java.
2. Use of Atomic Wrapper classes from java.util.concurrent.atomic package. For example AtomicInteger
3. Use of locks from java.util.concurrent.locks package.
4. Using thread safe collection classes, check this post for usage of ConcurrentHashMap for thread safety.
5. Using volatile keyword with variables to make every thread read the data from memory, not read from thread cache.

Synchronized(this) - current object lock
Synchronized(Object obj) -- obj level lock
Synchronized(Object.class) -- class level lock

class level lock - if thread will get class lock other threads are not allowed to execute other static synchronized methods.

Executor services

ExecutorService executorService1 = Executors.newSingleThreadExecutor();
>> Creates a ExecutorService object having a single thread.
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
>> Creates a ExecutorService object having a pool of 10 threads.
>> number of thread is depend on core counts or in case of io ops number of/avg of tasks .
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
>> Creates a scheduled thread pool executor

Imp Methods

execute(Runnable task) -defined in executor interface
submit(Runnable task) / submit(Callable<T> task) -defined in executerService interface

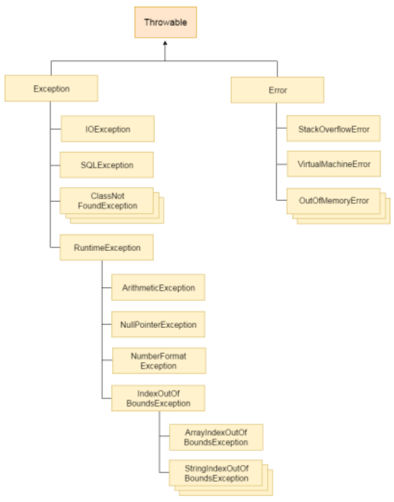
invokeAny(Collection<? extends Callable<T>> tasks)
invokeAll(Collection<? extends Callable<T>> tasks)

Once we are done with our tasks given to ExecutorService, then we have to shut it down because ExecutorService performs the task on different threads. If we don't shut down the ExecutorService, the threads will keep running, and the JVM won?t shut down.
The process of shutting down can be done by the following three methods-

shutdown() method   // initiate shutdown + not accept new task + complete existing task
shutdownNow() method // initiate shutdown + complted task which are running + return list of task which are queued
awaitTermination(10 , TimeUnit.SECOND) method // block termination untill all tasks completed or timeout

isShutdown() //return true if shutdown initiated
isTerminated() // returns true if all tasks are completed

Customized Exception :



```
class Test extends RuntimeException{

        //call super constructors
}


class Test extends Exception{

        //call super constructors
}
```

Handline Exceptions

1
```
try(FileInputStream fis=new FileInputStream("")){
        //can define without finally and catch block
}
```

2.
```
try {
        try {
            //execute business logic
        }catch (Exception e) {
                // TODO: handle exception
        }
}catch (Exception e) {
        // TODO: handle exception
}
```

3.
```
try {
        System.out.println("inside try");
        // System.exit(0);  // wont run finally
        I=10;
        return
}catch(Exception e) {
        System.out.println("inside catch");
}finally {
        I=20
        System.out.println("inside finally");
}
```

Output : inside try > inside finally > 10 | finally block value wont impact on returned value

The only times finally won't be called are:

If you invoke System.exit()
If you invoke Runtime.getRuntime().halt(exitStatus)
If the JVM crashes first
If the JVM reaches an infinite loop (or some other non-interruptable, non-terminating statement) in the try or catch block
If the OS forcibly terminates the JVM process; e.g., kill -9 <pid> on UNIX
If the host system dies; e.g., power failure, hardware error, OS panic, et cetera
If the finally block is going to be executed by a daemon thread and all other non-daemon threads exit before finally is called

How can you catch an exception thrown by another thread in Java

This can be done using Thread.UncaughtExceptionHandler.
Here’s a simple example:

```
// create our uncaught exception handler
Thread.UncaughtExceptionHandler handler = new Thread.UncaughtExceptionHandler() {
   public void uncaughtException(Thread th, Throwable ex) {
      System.out.println("Uncaught exception: " + ex);
   }
};

// create another thread
Thread otherThread = new Thread() {
   public void run() {
      System.out.println("Sleeping ...");
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {
         System.out.println("Interrupted.");
      }
      System.out.println("Throwing exception ...");
      throw new RuntimeException();
   }
};

// set our uncaught exception handler as the one to be used when the new thread
// throws an uncaught exception
otherThread.setUncaughtExceptionHandler(handler);

// start the other thread - our uncaught exception handler will be invoked when
// the other thread throws an uncaught exception
otherThread.start();
```

Class Loaders :

1. Bootstrap Classloader: Loads core java API file rt.jar from folder.
2. Extension Classloader: Loads jar files from folder.
3. System/Application Classloader: Loads jar files from path specified in the CLASSPATH environment variable.

Inner classes

1. Normal inner classes :

1. cant define static members/methods inside inner class but can access static members of outer class.
2. this.x -inner | outer.this.x - for outer

2. Method inner classes

3. Anonymous Inner classes

4. static inner classes

<u>Collection</u>

List :

ArrayList

LinkedList

Vector ← Stack

Set :

HashSet

LinkedHashSet

SortedSet(I) ← NavigableSet(I) ← TreeSet

Queue :

PriorityQueue

Dequeue ← ArrayDequeu

Map :

HashMap

LinkedHashMap

WeekHashMap

IdentityHashMap

SortedMap(I) ←NaviagableMap (I) ← TreeMap

ArrayList :

ArrayList() : This constructor is used to create an empty ArrayList with an initial capacity of 10 and this is a default constructor. We can create an empty Array list by reference name arr name object of ArrayList class as shown below.

```
ArrayList arr_name = new ArrayList();
ArrayList arr_name = new ArrayList(int capacity);
ArrayList arr_name = new ArrayList(Colelction c);
```

Growable Nature :

For example, if the Array size is 10 and already all the rooms were filled by the elements, while we are adding a new element now the array capacity will be increased as 10+ (10>>1) => 10+ 5 => 15. Here the size is increased from 10 to 15. So increase the size by 50% we use right shift operator.

Threshold :

Threshold = (Current Capacity) * (Load Factor)

The load factor is the measure that decides when to increase the capacity of the ArrayList. The default load factor of an ArrayList is 0.75f. For example, current capacity is 10. So, loadfactor = 10*0.75=7 while adding the 7th element array size will increase.

Hashset :

HashSet, it internally creates a HashMap and if we insert an element into this HashSet using add() method, it actually call put() method on internally created HashMap object with element you have specified as it's key and constant Object called "PRESENT" as it's value. So we can say that a Set achieves uniqueness internally through HashMap. Now the whole story comes around how a HashMap and put() method internally works.

```
 // A Dummy value(PRESENT) to associate with an Object in the Map
                private static final Object PRESENT = new Object();
 // default constructor of HashSet class  , It creates a HashMap by calling  ,  default constructor of HashMap class
                 public HashSet() {
                    map = new HashMap<>();
                 }
  // add method ,  it calls put() method on map object and then compares it's return value with null
               public boolean add(E e) {
                        return map.put(e, PRESENT)==null;
               }
```

HashMap

HashMap hm=new HashMap(int initialCapacity, float loadFactor);

The initial capacity of the HashMap is the number of buckets in the hash table. It creates when we create the object of HashMap class. The initial capacity of the HashMap is 24, i.e., 16. The capacity of the HashMap is doubled each time it reaches the threshold. The capacity is increased to 25=32, 26=64, and so on.

Load Factor

The Load factor is a measure that decides when to increase the HashMap capacity to maintain the get() and put() operation complexity of O(1). The default load factor of HashMap is 0.75f (75% of the map size).

- initial capacity of the hashmap*Load factor of the hashmap.
- The initial capacity of hashmap is=16
- The default load factor of hashmap=0.75
- According to the formula as mentioned above: 16*0.75=12

It represents that 12th key-value pair of hashmap will keep its size to 16. As soon as 13th element (key-value pair) will come into the Hashmap, it will increase its size from default 16 buckets to 32 buckets.

What’s wrong using HashMap in the multi-threaded environment? When get() method go to the infinite loop?

Well, nothing is wrong, it depending upon how you use. For example, if you initialize the HashMap by just one thread and then all threads are only reading from it, then it's perfectly fine. **One example of this is a Map which contains configuration properties. The real problem starts when at least one of that thread is updating HashMap** i.e. adding, changing or removing any key value pair. Since put() operation can cause re-sizing and which can further lead to infinite loop, that's why either you should use Hashtable or ConcurrentHashMap, later is better.

Does not overriding hashCode() method has any performance implication?

This is a good question and open to all, as per my knowledge a poor hashcode function will result in the frequent collision in HashMap which eventually increase the time for adding an object into HashMap. From Java 8 onwards though collision will not impact performance as much as it does in earlier versions because after a threshold the linked list will be replaced by the binary tree, which will give you O(logN) performance in the worst case as compared to O(n) of linked list.

There are several differences between HashMap and Hashtable in Java:

Hashtable is synchronized, whereas HashMap is not. This makes HashMap better for non-threaded applications, as unsynchronized Objects typically perform better than synchronized ones.
Hashtable does not allow null keys or values. HashMap allows one null key and any number of null values.
One of HashMap's subclasses is LinkedHashMap, so in the event that you'd want predictable iteration order (which is insertion order by default), you could easily swap out the HashMap for a LinkedHashMap. This wouldn't be as easy if you were using Hashtable.

Priority Queue:

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. The PriorityQueue is based on the priority heap.
The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

Let's understand the priority queue with an example:
// Java program to demonstrate the working of priority queue in Java

```
import java.util.*;
 class GfG {
   public static void main(String args[])
   {
      // Creating empty priority queue
      PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();

      // Adding items to the pQueue using add()
      pQueue.add(10);
      pQueue.add(20);
      pQueue.add(15);


     // Printing the top element of PriorityQueue
      System.out.println(pQueue.peek());

      // Printing the top element and removing It from the PriorityQueue container
      System.out.println(pQueue.poll());


      // Printing the top element again
      System.out.println(pQueue.peek());
   }
}
```

Output:

```
10 –peek element
10 –poll element
15 -peek element
```

ArrayDeque **:**

ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue. Array deques have no capacity restrictions and they grow as necessary to support usage.
Lets understand ArrayDeque with an example:

```
import java.util.*;
public class ArrayDequeDemo {
   public static void main(String[] args)
   {
      // Initializing an deque
      ArrayDeque<Integer> de_que = new ArrayDeque<Integer>(10);

      // add() method to insert
      de_que.add(10);
      de_que.add(20);
      de_que.add(30);
      de_que.add(40);
      de_que.add(50);

      System.out.println(de_que);

      // clear() method
      de_que.clear();

      // addFirst() method to insert the
      // elements at the head
      de_que.addFirst(564);
      de_que.addFirst(291);

      // addLast() method to insert the
      // elements at the tail
      de_que.addLast(24);
      de_que.addLast(14);

      System.out.println(de_que);
   }
}
```

Output:

```
[10, 20, 30, 40, 50]
[291, 564, 24, 14]
```

File Operations

FileWriter -- no line seperator available
BufferedWriter -- line seperator available

FileReader -- reads data char by char
BufferedFileReader -- reads data line by line

PrintWriter -- most enhanced file writter , can write data in any datatype to file
ex println(int i) , println(double d) , etc

Cloaning : 1. shallow cloaning **2**. deep cloaning

#ShallowCloning : creating reference of the object called shallow cloning

```
Test t1=new Test();
Test t2=t1;

class Test2 implements Cloneable {
   int a;
   int b;
   Test c = new Test();
   public Object clone() throws CloneNotSupportedException
   {
      return super.clone();
   }
}
```

#DeepCloninig : Creating duplicate object

```
          @Override
   protected Object clone() throws CloneNotSupportedException {

          Test3 test3=(Test3) super.clone();
          test3.c=new Test();
          test3.c.x=c.x;
          test3.c.y=c.y;
          return test3;

   }
```

JDBC

```
Class.forName("oracle.jdbc.driver.OracleDriver");

Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");

PreparedStatement stmt=con.prepareStatement("insert into Emp values(?,?)");
stmt.setInt(1,101);//1 specifies the first parameter in the query
stmt.setString(2,"Ratan");

int i=stmt.executeUpdate();
```

Java Concurrency

Volatile vs Automic Integer

Volatile -- > data pushed to local cache --> data pushed to shared cache
* Generally, use for flags
*Not recommended for counter operations

Automic Data Type :
AtomicInteger / AtomicBoolean /AtomicLong
*Thread safe data type , where volatile gets failed in synchronization while counter operation atomic data types does the job.

```
AtomicInteger integer=new AtomicInteger(5);
System.out.println(integer.incrementAndGet());
```

ThreadLocal
The ThreadLocal class is used to create thread local variables which can only be read and written by the same thread.

```
ThreadLocal<Integer> threadLocalCounter = new ThreadLocal<Integer>();
```

Mathods :
1.public T get()
2.protected T initialValue()
3.public void remove()
4.public void set(T value)

<u>ReentrantLock</u>

```
ReentrantLock lock = new ReentrantLock();
public void lockDemo() {
        lock.lock();
        try {
                lock.tryLock(1000, TimeUnit.SECONDS);
        } catch (Exception e) {
                // TODO: handle exception
        } finally {
                lock.unlock();
        }
}
```

<u>ReentrantReadWriteLock</u>
for read lock multiple thread can acquire lock whereas for write operation only single thread can acquire lock.

```
ReentrantReadWriteLock.ReadLock readLock=new ReentrantReadWriteLock().readLock();
ReentrantReadWriteLock.WriteLock writeLock=new ReentrantReadWriteLock().writeLock();

public void readLock() {
        readLock.lock();
        //heavy operation
        readLock.unlock();
}
public void writeLock() {
        writeLock.lock();
        //heavy operation
        writeLock.unlock();
}
```

<u>Semaphore</u>

Semaphore is to provide permits to thread to access service. only defined number of threads can access services at a time.

```
Semaphore semaphore=new Semaphore(3); //only 3 thread to access service
public void heavyDuty() {
        //semaphore.acquire(); // thrown interrupted exception
        semaphore.acquireUninterruptibly();

        //heavy operations
        semaphore.release();
}
```

<u>Conditional Class</u>

<u>CountDown Latch & Cyclic Barrier</u>

CountDownLatch is a thread waiting for multiple threads to finish or calling countDown().
When all threads have called countDown(), the awaiting thread continues to execute.

CyclicBarrier is that different threads hang tight for one another(wait for each other)and when all have finished their execution, the result needs to be combined in the parent thread.

https://www.geeksforgeeks.org/difference-between-countdownlatch-and-cyclicbarrier-in-java/

Concurrent Collection

1. ConcurrentHashMap

```
ConcurrentHashMap m = new ConcurrentHashMap(200 , 0.75f, 10); //10 is concurrency level
```

Default capacity -- 16
null is not allowed for both key and value
Segment level is decided on the basis of concurrency level .
2,4,8,16,32

*Rehashing after load factor impacted

https://javabypatel.blogspot.com/2016/09/concurrenthashmap-interview-questions.html

2. CopyOnWriteArrayList
3. CopyOnWriteArraySet

-for every write operation clone will be created and operation will be performed on clone operation.
-- jvm will take care of sync operation of these data

BlockingQueue
----
can handle concurrent operation.
it is recommended for consumer producer problem

The BlockingQueue interface in Java is added in Java 1.5 along with various other concurrent Utility classes

BlockingQueue does not accept a null value. If we try to enqueue the null item, then it throws NullPointerException.
Java provides several BlockingQueue implementations such as LinkedBlockingQueue, ArrayBlockingQueue, PriorityBlockingQueue

A thread trying to enqueue an element in a full queue is blocked until some other thread makes space in the queue, either by dequeuing one or more elements or clearing the queue completely.
Similarly, it blocks a thread trying to delete from an empty queue until some other treads insert an item.

Java Reflection

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

Working of Garbage collection
https://www.youtube.com/watch?v=UnaNQgzw4zY

```
                                                    Heap
-------------------------------------------------------------------------------------------------------------------
younge Generation                                                                              Old generation

          Eden space
serviver 1                    serviver 2

-------------------------------------------------------------------------------------------------------------------
```

after triggering a perticuer threshol objects moved to old generation.
servivor-1 reachable object shifts to servivor -2
servivor -2 shofts to servivor -1

Serialization

Serialization in Java is a mechanism of writing the state of an object into a byte-stream
For serializing the object, we call the writeObject() method ObjectOutputStream, and for deserialization we call the readObject() method of ObjectInputStream class.

We must have to implement the Serializable interface for serializing the object.

```
public class Student implements Serializable{
 int id;
 String name;
 public Student(int id, String name) {
  this.id = id;
  this.name = name;
 }
}
```

java.io.Serializable interface
Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The Cloneable and Remote are also marker interfaces.

The String class and all the wrapper classes implement the java.io.Serializable interface by default.

Serielizable Interface
FileInputStream   ObjectInputStream
FileOutputStream   ObjectOutputStream

```
transient Striing name="sss";  // name=null will be stored
final transient Striing name="sss";  // name=sss will be stored  -- no impact of transient
 static transient Striing name="sss";  // name=Sss will be stored -- no impact of transient
```

Externalization

externalization
==================
transient wont work with externalization

Externalizable interface present in java.io, is used for Externalization which extends Serializable interface. It consist of two methods which we have to override to write/read object into/from stream which are-

```
// to read object from stream
void readExternal(ObjectInput in)

// to write object into stream
void writeExternal(ObjectOutput out)
```

Java 8 feature :

1. Lambda Expression :

```
        interface Cook{
                public void cookFood();
        }
        main(){
                Cook cook2=() ->{System.out.println("Cooking food");    };
        }
```

1. forEach() method in Iterable interface

```
                List<Integer> list= List.of(1,2,3,4,6,4,3,7,2,77);
                list.forEach(new Consumer<Integer>() {
                        @Override
                        public void accept(Integer t) {
                                System.out.print(t+" ");
                        }
                });
```

2. default and static methods in Interfaces

3. Functional Interface

```
                interface  Integrface1{
                        void print();
                        default void printData() {
                                System.out.println("Default methods");
                        }
                        static void logData() {
                                System.out.println("Static Method");
                        }
                        public static void main(String[] args) {
                                System.out.println("main method");
                        }
                }
                public class DefaultAndStaticMethods {
                        public static void main(String[] args) {
                                Integrface1.logData();
                                Integrface1.main(null);
                                //version 1
                                Integrface1 i1=new Integrface1() {
                                        @Override
                                        public void print() {
                                                System.out.println("print method ");
                                        }
                                };
                                i1.print();
                                //version 2
                                Integrface1 i2=()-> {System.out.println("print method ");};
                                i2.print();
                                //version 3
                                Integrface1 i3=()-> System.out.println("print method ");
                                i3.print();
                        }
                }
```

4. Stream API :
it allows functional-style operations on the elements. It performs lazy computation. So, it executes only when it requires.

- Sequential Stream
- Parallel Stream

Sequential Stream	**Parallel Stream**
Runs on a single-core of the computer	Utilize the multiple cores of the computer.
Performance is poor	The performance is high.
Order is maintained	Doesn't care about the order
Only a single iteration at a time just like the for-loop.	Operates multiple iterations simultaneously in different available cores.
Each iteration waits for currently running one to finish,	Waits only if no cores are free or available at a given time
More reliable and less error	Less reliable and error-prone.
Platform independent	Platform dependent

```
                ArrayList<Integer> list=new ArrayList<>();
                for (int i = 0; i < 50; i++)
                list.add(i);
                Stream<Integer> parallelStream=list.stream().parallel();
                Stream<Integer> sequentialStream=list.stream().sequential();
                System.out.println();
                System.out.println("SortedLIst");
ArrayList<Integer> sortedList=(ArrayList<Integer>) parallelStream.filter(p->p%2==0).collect(Collectors.toList());
                sortedList.forEach(t->System.out.print(t+" "));
                //once data published , streams will get empty
```

5 . Date & Time API

6. Optional Class

It is a public final class and used to deal with NullPointerException in Java application.

```
        import java.util.Optional;
        public class OptionalExample {
          public static void main(String[] args) {
            String[] str = new String[10];
            Optional<String> checkNull = Optional.ofNullable(str[5]);
            if(checkNull.isPresent()){  // check for value is present or not
              String lowercaseString = str[5].toLowerCase();
              System.out.print(lowercaseString);
            }else
              System.out.println("string value is not present");
          }
        }
```

7. Method Reference

a. Reference to a static method.
b. Reference to an instance method.
c. Reference to a constructor.

```
interface Message{
	void printMessage();
}

class Talk{

	public Talk() {
		System.out.println("Constructor is talking to you");
	}

	public void talk() {
		System.out.println("talking to you");
	}
	public static void talk2() {
		System.out.println("talking to you");
	}
}
public class MethodReference {
	public static void main(String[] args) {

		//static refrence
		Message msg1=Talk::talk2;
		msg1.printMessage();

		//instance Method Refrence
		Message msg2=new Talk()::talk;
		msg2.printMessage();

		//constructor reference
		Message msg3=Talk::new;
		msg3.printMessage();
	}
}
```

* Hashmap Enhancement

Java 9 Features

1. **private method in interface**

```
public interface Card {
        private Long createCardID(){
                // Method implementation goes here.
        }
        private static void displayCardDetails(){
                // Method implementation goes here.
        }
}
```

2. **Try-With Resources Enhancement**

```
        //Defining resources outside of try(----) is allowed
        FileOutputStream fileStream=new FileOutputStream("javatpoint.txt");
         try(fileStream){
                 String greeting = "Welcome to javaTpoint.";
                  byte b[] = greeting.getBytes();
                  fileStream.write(b);
                  System.out.println("File written");
           }catch(Exception e) {
                 System.out.println(e);
           }
```

2. Modularization

1. Required module-info.java file

```
module com.amazonprime {
        exports com.ad.prime;
}
```

2. Exports package; | requires module;

```
module com.ott {
        requires com.amazonprime;
        requires com.youtube;
}
```

3. Add dependedn module in builpath -> project -> module - > add
4. Example :

    module can be - AmazonPrime , Youtube
    Main Module --> AccessOtt

Java 10 Features

1. Type variable inference

```
var numbers = List.of(1, 2, 3, 4, 5); // inferred value ArrayList<String>

// Index of Enhanced For Loop
for (var number : numbers) {
        System.out.println(number);
}

// Local variable declared in a loop
for (var i = 0; i < numbers.size(); i++) {
        System.out.println(numbers.get(i));
}
```

Design Pattern

Singletone

```
public class Singleton {
    private static volatile Singleton _instance;
            /**
             * Double checked locking code on Singleton
             * @return Singelton instance
             */
    public static Singleton getInstance() {
        if (_instance == null) {
            synchronized (Singleton.class) {
                if (_instance == null) {
                    _instance = new Singleton();
                }
            }
        }
        return _instance;
    }
}
```

Factory

Adaptor

Decorator

Abstract Class	Interfaces
An abstract class can provide complete, default code and/or just the details that have to be overridden	An interface cannot provide any code at all, just the signature
In the case of an abstract class, a class may extend only one abstract class	A Class may implement several interfaces
An abstract class can have non-abstract methods	All methods of an Interface are abstract
An abstract class can have instance variables	An Interface cannot have instance variables
An abstract class can have any visibility: public, private, protected	An Interface visibility must be public (or) none
If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method
An abstract class can contain constructors	An Interface cannot contain constructors
Abstract classes are fast	Interfaces are slow as it requires extra indirection to find the corresponding method in the actual class

Spring

Spring boot

Spring Boot is a module of Spring Framework. It allows us to build a stand-alone application with minimal or zero configurations. It is better to use if we want to develop a simple Spring-based application or RESTful services.

Spring Boot is a project that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run both simple and web-based applications.

It is a Spring module that provides the RAD (Rapid Application Development) feature to the Spring Framework.
It is used to create a stand-alone Spring-based application that you can just run because it needs minimal Spring configuration.

The dependency injection approach is used in Spring Boot. It contains powerful database transaction management capabilities. It simplifies integration with other Java frameworks like JPA/Hibernate ORM, Struts, etc. It reduces the cost and development time of the application.

<u>ways to create spring boot application</u>
=======================================
1. Spring initializer
2. maven

spring-boot-starter-parent
spring-boot-starter
spring-boot-starter-test

maven
============

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>2.2.1.RELEASE</version>
<type>pom</type>
</dependency>

<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
<version>2.2.1.RELEASE</version>
</dependency>
<dependency>
```

Main Class – Spring Boot

```java
@SpringBootApplication
@ComponentScan(basePackages = "com.myapp")
public class MyappApplication {

	public static void main(String[] args) {
		SpringApplication.run(MyappApplication.class, args);
	}

}
```

SpringBootApplication vs EnableAutoConfiguration

@EnableAutoConfiguration: It auto-configures the bean that is present in the class path and configures it to run the methods. The use of this annotation is reduced in Spring Boot 1.2.0 release because developers provided an alternative of the annotation, i.e. @SpringBootApplication.

@SpringBootApplication: It is a combination of three annotations @EnableAutoConfiguration, @ComponentScan, and @Configuration. [@SpringBootApplication=@ComponentScan+@EnableAutoConfiguration+@Configuration]

Excluding classes from execution
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class}) release version 1.0 (componentscan + configuration)

@SpringBootApplication (exclude={JacksonAutoConfiguration.class, JmxAutoConfiguration.class})

OR add the following statement in the application.properties file.
spring.autoconfiguration.exclude=org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration

Manual Configuration
Spring Boot Framework comes with a built-in mechanism for application configuration using a file called application.properties.

```
application.properties
------------------------------------------------
spring.application.name = demoApplication
server.port = 8081
server.ssl.enabled=true
```

Spring Boot DevTools

Spring Boot 1.3 provides another module called Spring Boot DevTools. DevTools stands for Developer Tool. Spring Boot DevTools pick up the new changes and restart the application automatically.

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <scope>runtime</scope>
      <optional>true</optional>
</dependency>
```

Spring boot version 2.2.4

```java
@RestController
public class StudentController {

	@Autowired
	StudentService studentService;

	@GetMapping(path = "/get/{id}")
	public @ResponseBody Student getStudent(@PathVariable int id) {
		return studentService.getStudent(id);
	}

	@PostMapping(value = "/create")
	public Student create(@RequestBody Student student) {
		return studentService.add(student);
	}

	@GetMapping("/getall")
	public List<Student> getAll() {
		return studentService.getStudentList();
	}

	@DeleteMapping("/delete/{id}")
	public List<Student> remove(@PathVariable int id) throws Exception {
		System.out.println("id : "+id);
		return studentService.remove(id);
	}

}
```

@GetMapping: It maps the **HTTP GET** requests on the specific handler method. It is used instead of using : **@RequestMapping(method = RequestMethod.GET)**

- **@PostMapping:** It maps the **HTTP POST** requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.POST)**
- **@PutMapping:** It maps the **HTTP PUT** requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PUT)**
- **@DeleteMapping:** It maps the **HTTP DELETE** requests on the specific handler method It is used instead of using: **@RequestMapping(method = RequestMethod.DELETE)**
- **@PatchMapping:** It maps the **HTTP PATCH** requests on the specific handler method. It is used instead of using: **@RequestMapping(method = RequestMethod.PATCH)**
- **@RequestBody:** It is used to **bind** HTTP request with an object in a method parameter. When we annotate a method parameter with **@RequestBody,** the Spring framework binds the incoming HTTP request body to that parameter.
- **@ResponseBody:** It binds the method return value to the response body. It tells the Spring Boot Framework to serialize a return an object into JSON and XML format.
- **@PathVariable:** It is used to extract the values from the URI. It is most suitable for the RESTful web service, where the URL contains a path variable. We can define multiple @PathVariable in a method. Ex : localhost:8080/get/3

- **@RequestParam:** It is used to extract the query parameters form the URL. It is also known as a **query parameter**. It is most suitable for web applications. It can specify default values if the query parameter is not present in the URL. Ex : localhost:8008/get?name=amol
- **@RequestHeader:** It is used to get the details about the HTTP request headers. We use this annotation as a **method parameter**. The optional elements of the annotation are **name, required, value, defaultValue.** For each detail in the header, we should specify separate annotations. We can use it multiple time in a method
- **@RestController:** It can be considered as a combination of **@Controller** and **@ResponseBody** annotations**.** The @RestController annotation is itself annotated with the @ResponseBody annotation. It eliminates the need for annotating each method with @ResponseBody.
- **@RequestAttribute:** It binds a method parameter to request attribute. It provides convenient access to the request attributes from a controller method. With the help of @RequestAttribute annotation, we can access objects that are populated on the server-side.

Exception Handling

```
@ControllerAdvice
public class MyAppExceptionHandler extends ResponseEntityExceptionHandler{


	@ExceptionHandler(value = Exception.class)
	public ResponseEntity<Object> handleAllException(Exception ex,WebRequest request) {
		ExceptionDescriiptor ed=new ExceptionDescriiptor(ex.toString(), request.getDescription(false),
		new Date().~~toLocaleString~~());
		return new ResponseEntity<Object>(ed, HttpStatus.BAD_REQUEST);
	}

	// can have multiple methods to handle exception
	// ResponseEntityExceptionHandler - handles spring mvc exception
}
```

Custom Class to build message

```
public class ExceptionDescriiptor {
	String exception;
	String requestFailureDetails;
	String timeStamp;


	public ExceptionDescriiptor(String exception, String requestFailureDetails, String timeStamp) {
		super();
		this.exception = exception;
		this.requestFailureDetails = requestFailureDetails;
		this.timeStamp = timeStamp;
	}

	//getter setter

}
```

Spring boot security

```
	<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-security</artifactId>
		</dependency>
```

Default User : user

Default password : Using generated security password: 2f04b294-9def-4763-96d7-77238777b5c9

application.properties

```
spring.security.user.name=amold
spring.security.user.password=amold@123
```

IMP

Getters are required to convert bean into rendering format (json/xml)

Validation API

```
                    <dependency>
                              <groupId>org.springframework.boot</groupId>
                              <artifactId>spring-boot-starter-validation</artifactId>
                    </dependency>
```

```
public Student create(@Valid @RequestBody Student student) {
                    return studentService.add(student);               }

@Range(max = 100)
private int id;

@Size(min = 2,message = "Name size should be greater than 2 characters")
private String name;
```

HATEOAS - hypermedia as the engine of application state

New changes : ResourceSupport is now RepresentationModel
Resource is now EntityModel
Resources is now CollectionModel
PagedResources is now PagedModel

package : org.springframework.hateoas

old version: Resource<Student> resource=new Resource<>(student);

New Version:

```
                    EntityModel<Student> resource=EntityModel.of(student);
                    resource.add(linkTo(methodOn(StudentController.class).getAll()).withRel("All-Students"));
                    resource.add(linkTo(methodOn(StudentController.class).getAll()).withRel("All-Students"));


          @GetMapping(path = "/get/{id}")
          public @ResponseBody EntityModel<Student> getStudent(@PathVariable int id) throws Exception {

                    Student student=studentService.getStudent(id);
                    if(null==student)
                              throw new Exception("Student Not Found");


                    EntityModel<Student> resource=EntityModel.of(student);
                    resource.add(linkTo(methodOn(StudentController.class).getAll()).withRel("All-Students"));
                    resource.add(linkTo(methodOn(StudentController.class).getAll()).withRel("All-Students"));
                    return resource;
          }
```

Actuator : <u>Service Monitoring Tool</u>

```
                    <dependency>
                              <groupId>org.springframework.boot</groupId>
                              <artifactId>spring-boot-starter-actuator</artifactId>
                    </dependency>


                    http://localhost:8080/actuator
```

Hibernate

hibernate.cfg.xml

<property name="hbm2ddl.auto">create<property> //create schema every time

<property name="hbm2ddl.auto">update<property> //create schema first time then update it for later iterations

<mapping class="com.ad.ABC"> // mapping of each entity , if mapping is not present it wont get persisted

some important interfaces of Hibernate framework :

SessionFactory (org.hibernate.SessionFactory): SessionFactory is an immutable thread-safe cache of compiled mappings for a single database. We need to initialize SessionFactory once and then we can cache and reuse it. SessionFactory instance is used to get the Session objects for database operations.

Session (org.hibernate.Session): Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps JDBC java.sql.Connection and works as a factory for org.hibernate.Transaction. We should open session only when it's required and close it as soon as we are done using it. Session object is the interface between java application code and hibernate framework and provide methods for CRUD operations.

Transaction (org.hibernate.Transaction): Transaction is a single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC or JTA transaction. A org.hibernate.Session might span multiple org.hibernate.Transaction in some cases.

Hibernate SessionFactory is thread safe?

Internal state of SessionFactory is immutable, so it's thread safe. Multiple threads can access it simultaneously to get Session instances.

Hibernate Session is thread safe?

Hibernate Session object is not thread safe, every thread should get it's own session instance and close it after it's work is finished.

What are different states of an entity bean?

An entity bean instance can exist is one of the three states.

Transient: When an object is never persisted or associated with any session, it's in transient state. Transient instances may be made persistent by calling save(), persist() or saveOrUpdate(). Persistent instances may be made transient by calling delete().

Persistent: When an object is associated with a unique session, it's in persistent state. Any instance returned by a get() or load() method is persistent.

Detached: When an object is previously persistent but not associated with any session, it's in detached state. Detached instances may be made persistent by calling update(), saveOrUpdate(), lock() or replicate(). The state of a transient or detached instance may also be made persistent as a new persistent instance by calling merge().

What is difference between Hibernate Session get() and load() method?

Hibernate session comes with different methods to load data from database. get and load are most used methods, at first look they seems similar but there are some differences between them.

get() loads the data as soon as it's called whereas load() returns a proxy object and loads data only when it's actually required, so **load() is better because it support lazy loading**.

Since **load() throws exception when data is not found**, we should use it only when we know data exists.
We should use get() when we want to make sure data exists in the database. **get() returns null if data not found**

What is hibernate caching :

Query Cache

As the name suggests, hibernate caches query data to make our application faster. Hibernate Cache can be very useful in gaining fast application performance if used correctly. The idea behind cache is to reduce the number of database queries, hence reducing the throughput time of the application.

```
<property name="hibernate.cache.use_query_cache">true</property>
```

```
Query query = session.createQuery("from Employee");
query.setCacheable(true); // add setCacheable(true) in code to cache query
query.setCacheRegion("ALL_EMP");
```

First level cache

Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However, hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.

Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.

Second Level Cache using EHCache

EHCache is the best choice for utilizing hibernate second level cache.
Following steps are required to enable EHCache in hibernate application.

Add hibernate-ehcache dependency in your maven project, if it's not maven then add corresponding jars.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>4.3.5.Final</version>
</dependency>
```

Add below properties **in hibernate configuration** file.

```
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>
<property name="hibernate.cache.use_second_level_cache">true</property>
```

Annotate entity beans with **@Cache** annotation and caching strategy to use. For example,

```
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;
@Entity
@Table(name = "ADDRESS")
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY, region="employee")
public class Address {

}.
```

Sample Program

```
@Entity  // marks class as entity to persist into table
@Table //table name same as class name
@Table(name= "table_1") //custom table name
class ABC{

        @Id
        @GeneratedValue(strategy=GenerationType.Auto) //GenerationType.sequence // GenerationType.Identity
        int id;

        @Temporal(TemporalType.Date) //save only date and ignores time
        Date joinedDate;

        @LOB  // treat it as large object
        String discription;

        @Embedded // embedd one object in this object
        Address address;

        @Embedded
        @AttributeOverrides{ // use to override column names with default
                @attributeoverride(name="city" column=@column(name="Home_city"))
        }
        Address homeAddress;

        @ElementCollection  // use to store collection data into table
        @joinTable(name="user_address" , joincolumn=@joincolumn(name="userID"))
        /*non mandatory annotation, provides custom name to table and foreign key
         by default tablename_variablename is table name          */
        HashSet<Address> listOfAddress=new HashSet<>();

}

//value object
@Embeddable  // use to persist object within object
class Address{
        @Coulumn(name="city_name")
        String city;

        @Coulumn(name="pincode_number")
        String picode;
}
```

Using Hibernate Components :

```
 SessionFactory sessionfaactory=new Configuration().configure().buildSessionFactory();
Session session=sessionfaactory.openSession();
Transaction transaction=session.beginTransaction();
session.save(new ABC());
transaction.commit();
```

Mapping

1. one to one
2. Many to one / one to many
3. Many to many

```
class Person{

        @oneToOne
        Adharcard adharCard;


        @OneToMany
        ArrayList<Belt> belt=new ArrayList<>();
        //save all the belt object
}


class AdharCard{

}


class Belt{

        int id;
        String name;

        @ManyToOne
        Person person;
}


save(Person);
Save(AdharCard)
/*Save all the entities in the same order it used */


class vehical{

        @ManyToMany
        ArrayList <User> userList=new ArrayList<User>();
}



class User{

        @ManyToMany(mappedby="userList")
        ArrayList <vehical> vehicalList=new ArrayList<vehical>();
}
```

*mappedby ignores creation of redundant data table for many to many mapping.
*if mapped by is not used at any side then 2 separate tables will be created for mapping having same data.

Collections

Bag -- List/ArrayList unordered data
List --List/ArrayList ordered data
set
Map

Inheritance >> single inheritance

```
//@Inheritance(Strategy=inheritanceType.Single_Table) >>  default
@Inheritance(Strategy=inheritanceType.Table_Per_class) >> creates table per class
class Vehical{

}

class TwoWheelerVehical extends Vehical{

}
```

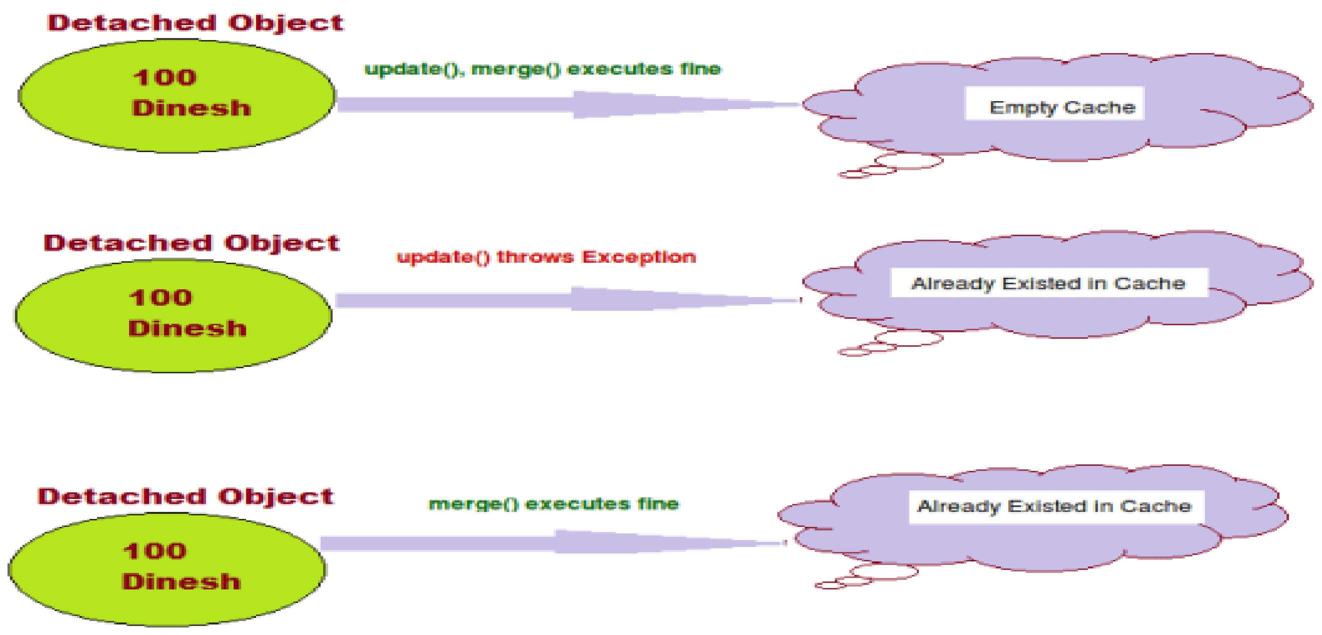
What is difference between Hibernate save(), saveOrUpdate() and persist() methods?

Hibernate save can be used to save entity to database. Problem with save() is that it can be invoked without a transaction and if we have mapping entities, then only the primary object gets saved causing data inconsistencies. Also **save returns the generated id immediately**.

Hibernate persist is similar to save with transaction. I feel it's better than save because we can't use it outside the boundary of transaction, so all the object mappings are preserved. Also **persist doesn't return the generated id immediately**, so data persistence happens when needed.

merge vs update

In the hibernate session we can maintain only one employee object in persistent state with same primary key, while converting a detached object into persistent, if already that session has a persistent object with the same primary key then hibernate throws an Exception whenever update() method is called to reattach a detached object with a session. In this case we need to call merge() method instead of update() so that hibernate copies the state changes from detached object into persistent object and we can say a detached object is converted into a persistent object.



How to implement Joins in Hibernate?

There are various ways to implement joins in hibernate.

>> Using associations such as one-to-one, one-to-many etc.
>> Using JOIN in the HQL query. There is another form "join fetch" to load associated data simultaneously, no lazy loading.
>> We can fire native sql query and use join keyword.

Native sql query in hibernate?

```
query = session.createSQLQuery("select e.emp_id, emp_name, emp_salary,address_line1, city,
            zipcode from Employee e, Address a where a.emp_id=e.emp_id");
rows = query.list();
```

How to log hibernate generated sql queries in log files?

We can set below property for hibernate configuration to log SQL queries.

```
    <property name="hibernate.show_sql">true</property>
```

How transaction management works in Hibernate?

Transaction management is very easy in hibernate because most of the operations are not permitted outside of a transaction. So after getting the session from **SessionFactory**, we can call session **beginTransaction**() to start the transaction. This method returns the Transaction reference that we can use later on to either commit or rollback the transaction.

Overall hibernate transaction management is better than JDBC transaction management because we don't need to rely on exceptions for rollback. Any exception thrown by session methods automatically rollback the transaction.

SQL

update query

```
UPDATE CUSTOMERS SET ADDRESS = 'Pune', SALARY = 1000.00 where id=10;
```

Order By

```
SELECT column-list  FROM table_name
[WHERE condition]
[ORDER BY column1, column2, .. columnN] [ASC | DESC];
```

Group By

```
SELECT column1, column2 FROM table_name
WHERE [ conditions ]
GROUP BY column1, column2
ORDER BY column1, column2
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Ramesh   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | kaushik  |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

```
SELECT NAME, SUM(SALARY) FROM CUSTOMERS GROUP BY NAME;
```

```
+---------+-------------+
| NAME    | SUM(SALARY) |
+---------+-------------+
| Hardik  |     8500.00 |
| kaushik |     8500.00 |
| Komal   |     4500.00 |
| Muffy   |    10000.00 |
| Ramesh  |     3500.00 |
+---------+-------------+
```

Data integrity is the overall accuracy, completeness, and consistency of data.

Joins
Inner join
left outer join
right outer join
full outer join

Example

```
  SELECT  ID, NAME, AMOUNT, DATE FROM CUSTOMERS
  LEFT/RIGHT/FULL JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

<u>self join:</u>
The SQL SELF JOIN is used to join a table to itself as if the table were two tables;

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_field = b.common_field;
```

<u>Cross join</u>
The CARTESIAN JOIN or CROSS JOIN returns the Cartesian product of the sets of records from two or more joined tables.

```
SELECT table1.column1, table2.column2...
FROM  table1, table2 [, table3 ]
```

<u>Union /union All</u>

The SQL UNION clause/operator is used to combine the results of two or more SELECT statements without returning any duplicate rows.

To use this UNION clause, each SELECT statement must have

- The same number of columns selected
- The same number of column expressions
- The same data type and
- Have them in the same order

<u>Example :</u>

```
SQL> SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS
  LEFT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID
UNION
  SELECT ID, NAME, AMOUNT, DATE FROM CUSTOMERS
  RIGHT JOIN ORDERS
  ON CUSTOMERS.ID = ORDERS.CUSTOMER_ID;
```

* The UNION ALL operator is used to combine the results of two SELECT statements including duplicate rows.

<u>Index</u>

Indexes are special lookup tables that the database search engine can use to speed up data retrieval.
Simply put, an index is a pointer to data in a table. An index in a database is very similar to an index in the back of a book.

1.CREATE INDEX index_name ON table_name;
2. Single index : CREATE INDEX index_name ON table_name (column_name);
3. Unique index : CREATE UNIQUE INDEX index_name on table_name (column_name);
/* **Unique indexes** are **indexes** that help maintain data integrity by ensuring that no two rows of data in a table have identical key values*/
4. Composite index : CREATE INDEX index_name on table_name (column1, column2);
5. DROP INDEX index_name;

Views

A view is nothing more than a SQL statement that is stored in the database with an associated name.

```
SQL > CREATE VIEW CUSTOMERS_VIEW AS  SELECT name, age FROM  CUSTOMERS;
SQL > UPDATE CUSTOMERS_VIEW          SET AGE = 35     WHERE name = 'Ramesh';
```

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. The same concept will apply on insert / delete

Having

The HAVING Clause enables you to specify conditions that filter which group results appear in the results

```
                    SELECT FROM WHERE
                    GROUP BY
                    HAVING
                    ORDER BY
```

```
 SQL > SELECT ID, NAME, AGE, ADDRESS, SALARY
       FROM CUSTOMERS
       GROUP BY age
       HAVING COUNT(age) >= 2;
```

ACID Properties

***Atomicity**
By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all.

***Consistency**
This means that integrity constraints must be maintained so that the database is consistent before and after the transaction.

***Isolation**
This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed.

***Durability**
This property ensures that once the transaction has completed execution,
the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory.

Trigger

A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs.

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3, Student.per = Student.total * 60 / 100;
```

Comparable	Comparator
1) Comparable provides a **single sorting sequence**. In other words, we can sort the collection on the basis of a single element such as id, name, and price.	The Comparator provides **multiple sorting sequences**. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc.
2) Comparable **affects the original class**, i.e., the actual class is modified.	Comparator **doesn't affect the original class**, i.e., the actual class is not modified.
3) Comparable provides **compareTo() method** to sort elements.	Comparator provides **compare() method** to sort elements.
4) Comparable is present in **java.lang** package.	A Comparator is present in the **java.util** package.
5) We can sort the list elements of Comparable type by **Collections.sort(List)** method.	We can sort the list elements of Comparator type by **Collections.sort(List, Comparator)** method.