Om Namah Sivaya!

Java: (Platform Independent) → Applications
- Standalone
- mobile
- Enterprise
- Web

↳ high level
→ robust
→ Object oriented
→ Secure programming lang.

Debug:
Break point: Break the execution of code at that point

Step over: After clicking step over that Break point step will execute.

Step into: After clicking the step into, we can go into the function (which actually doesn't contain any break point).

goto statement

operator overloading

→ If the class is not public, we can save the file name with different file name.
(It compiles with file name and at runtime it runs with class name)

JDK, JRE and JVM (These are platform dependent because the config. of OS is different)

JVM doesn't exist physically

JRE exist physically → provide Runtime environment (developing Java applications)

Command line arguments (CLI)

→ In Java, Command line arguments can be passed to a program when it is executed from the command line

Variables: → local (within the method)
→ Instance (inside class, outside method) — it doesn't get memory at compile time, it gets memory at runtime
→ Static

|| → Doesn't check the 2nd condition.
| → check the condition

Ternary operator:

(a < b) ? a : b

If condition true → a (1st part)
If condition false → b (2nd part)

OOPs Concept (main aim is to implement real world entities)

→ Object
→ class
→ Inheritance
→ polymorphism
→ Abstraction
→ Encapsulation

21 38

150483 29 476
8 30 125

Object: (It can be physical & logical)

→ An entity that has state and behaviour is known as object
→ An object is an instance of class.
→ An object contains address and takes up some memory.

(State, Behaviour, Identity)

new keyword
↳ Allocate memory at runtime.

Class:

→ Collection of objects is called class. It is logical entity.

→ Class doesn't consume any space

Inheritance: (→ Why multiple inheritance not supported in java? → To reduce the complexity & simplify the language)

→ When an "object" acquires/contains all the properties of the parent object, is called inheritance

→ It provides code reusability. & achieve runtime polymorphism.

IS-A → parent-child relationship.

HAS-A → Aggregation (entity reference)

polymorphism:

→ If one task is performed in different ways.

→ In java, we used method overloading/overriding. to achieve polymorphism,

→ Compile time (overload a static method in java)

→ Runtime (A a = new B();) → child class object

→ It can't be achieved by data members

→ reference variable of parent class functionality is known

Abstraction:

→ Hiding internal details and showing as abstraction, it is

Encapsulation:

→ Binding code & data together into a single unit are known as encapsulation.

Methods: predefined method (println(), equals(), max(), compareTo() etc.)
User-defined method.

Static method → The method that belongs to class.
↳ we can call it without creating an object

Instance method → we can call it with creating object.

Accessors (get) — It returns the value of private field

Mutators (set) — It sets the value of private field

Wrapper classes: (Permanance = primitives, frameworks = wrapper classes)
For primitives, we have wrapper classes
Eg: int → Integer extends Object class
char → Character
* primitive value to class object is called Boxing / Auto Boxing
* wrapper object to primitive type is called Unboxing / Auto Unboxing

Constructor: No arg constructor (Default constructor)
parameterized Constr
(It is used to initialize the object)

→ Class Name = method name
→ No return type
→ Cannot be abstract, static, final & Synchronized
→ Because Constructor is never inherited

Java copy Constructor:
→ By Constructor
Shallow (Abc obj1 = obj;)
Deep (Initializing seperately)
Cloning (Abc obj1 = obj.clone();)

(Constructor return current class instance, you cannot use return type get it returns value)

Static: → It is used for memory management mainly
→ static variable gets memory only once in the class area at the time of class loading.

(static methods can be overloaded)

Method overloading: (By changing no. of arguments / By changing the datatype)

→ If a class has multiple methods have same name but different in parameters is known as method overloading.

→ method overloading & type promotion:

→ Method overriding: (Runtime polymorphism) (Static methods can't be override) → Why because static methods belongs to class area, and an instance methods belongs to heap area

- If child class has the same method as parent class.
- The method must have the same name as in parent class.
- " " " " " parameters " "

Stored

Covariant Return type:

The return types of class are different when we override

Super keyword(): [It is used if parent child class have same fields]

→ It is used to refer immediate parent class object.

parent { class = white; }

child { super.color; }

Instance Initializer Block:

{ } ; called After super() keyword

Compiler calls in this way

```
Constructor(){
    super();
  {Instance Initializer block};
    Constructor();
}
```

final keyword: Is used to restrict the user.

→ If you make variable as final keyword = we can't change it.

→ If you make final method = we can't override it

→ If we make final class = we can't extend it.

(Runtime polymorphism can't be achieved by Data members.)

Early Binding

Static ← Dynamic Binding

When type of obj is determined at compile time

(Late Binding)

When type of obj is determined at runtime.

[If there is any private, static final methods in class]

Interfaces (achieve abstraction)

= "IS-A" relationship. ↳ multiple Inheritance.

- Compiler by default adds
  - "public static final" to the data members
  - & "abstract" to methods.

Class ↑extends Class

Interface ↑extends Interface

Interface ↑Implements class

(An interface which has no member is known as marker(or) tagged interface.)

Instanceof:

→ Null values → false

→ Downcasting is possible by using Instanceof.

Exception handling:

↳ Checked (IOExcept, SQLExceptn etc)

↳ Unchecked (Runtime Exceptn)

↳ Error (outofm

Shallow Copy:

```
Abc obj1 = obj;
```

Deep Copy:

```
Abc obj1 = new Abc();
obj1.i = obj.i;
obj1.j = obj.j;
```

clone: (marker interface)

① class implements Cloneable

② clone() override throws CloneNotSupportException

Collections:

(Sorting is not possible in Arraylist. It is possible in List)

↳ Collections.sort(values);
Collections.reverse(values);

Array List:

(only Integer) ArrayList<Integer> values = new ArrayList<Integer>();

↑ generics

Collections (Anything) LinkedList

Increases size by 100% (10, 20, 40, 80)

Vector<Integer> v = new Vector<Integer>();

v.add(ch);
v.remove(index);

(We can't contain Array list for primitive datatypes. It is required to use wrapper classes in Java)

↗ (a.add(1); a.add(2);

LinkedList vs ArrayList (Increases size by 50%.)

↓ ↓

Doubly LinkedList Dynamic Array (10, 15,

- Didn't take much time it we assign the value in middle.
- difficult to search the value.
- Time consuming when we add or remove an element in the arraylist
- Each to search the value

Comparator:

Comparator < Integer > com = new ComImpl();

Collections.sort (values, com);

Comparable: (?)

Set Interface:

(HashSet, LinkedHashSet, TreeSet)

↓ If doesn't maintain Insertion order

↓ It maintain Insertion order

Set < Integer > values = new HashSet < > ();

values.add ( );

for (int i : values)

sysо (i);

[→ In set we can't have duplicate elements].

→ It will not give the values in sequence

→ HashSet (unique elements) 3

→ LinkedHashSet (extends the HashSet (base) + (unique elements)

→ It maintains insertion order.

→ If we want values in ascending order then we will put TreeSet

↓ doesn't allow null element

a.pollFirst()

a.pollLast()

a.descendingSet()

Map Interface:

→ (Sorted Map)

→ HashMap → Stores data based on hashfunction (random order)

→ LinkedHashMap → (same order)

→ TreeMap → (Alphabetically sorted order)

HashMap<key, Value> obj = new HashMap<...>();

m.put→

for (Map.Entry map : m.entrySet())

Sysc(map.getKey() + " " + map.getValue());

- Remove()
- size()
- keySet()

map.containsKey("Indonesia"); → T/F

map.get("Indonesia") → get

LinkedList:

- Singly (One way traversal)
- Doubly (
- Circular singly (Last node connects to first node)
- Circular doubly

Stack:

→ It follows LIFO (Last In First Out)

Queue:

~~FIFO~~

→ Simple (Linear)

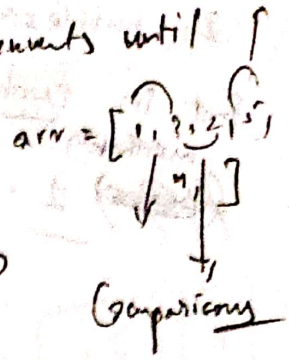
→ Circular (Last element connect to first) Ring Buffer.

→ Priority (Insertion FIFO, deletion priority) < Ascending / Descending

→ Double Ended queue.

Sorting techniques:

- **Bubble sort:** repeatedly swapping of adjacent elements until they are not in the intended order



- not suitable for large datasets

Worst Complexity → $O(n^2)$ | Best case → $O(n)$

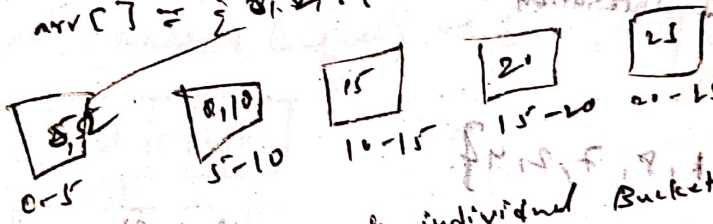
Avg

- Space Complexity → $O(1)$

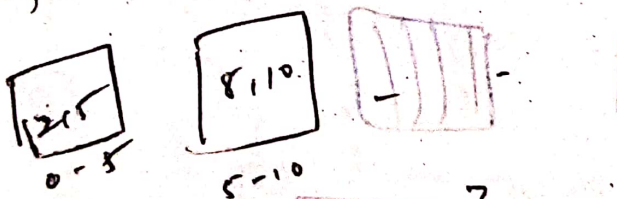
Bucket sort:

Each element inserted into the bucket

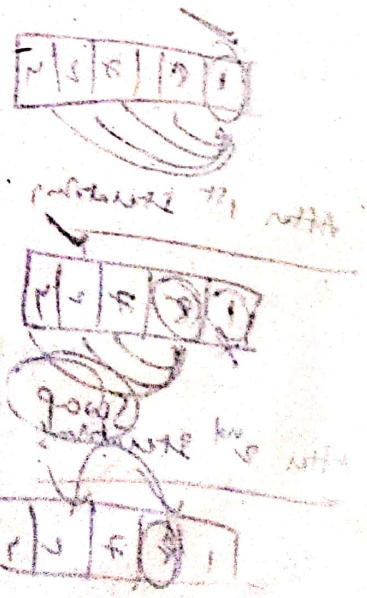
Suppose arr[] = {8, 10, 15, 13, 20}



→ Then we will sort each individual Bucket with some sorting techniques



arr[] = {2, 5, 8, 10, 20, 25}



Best Case → $O(n+k)$

& Avg

Worst → $O(n^2)$

Space Complexity → $O(n*k)$

Comb Sort: (Improvement Bubble sort)

↓

In bubble sort Comparison b/w elements were adjacent

But in combsort comparison gap b/w elements is more than 1.

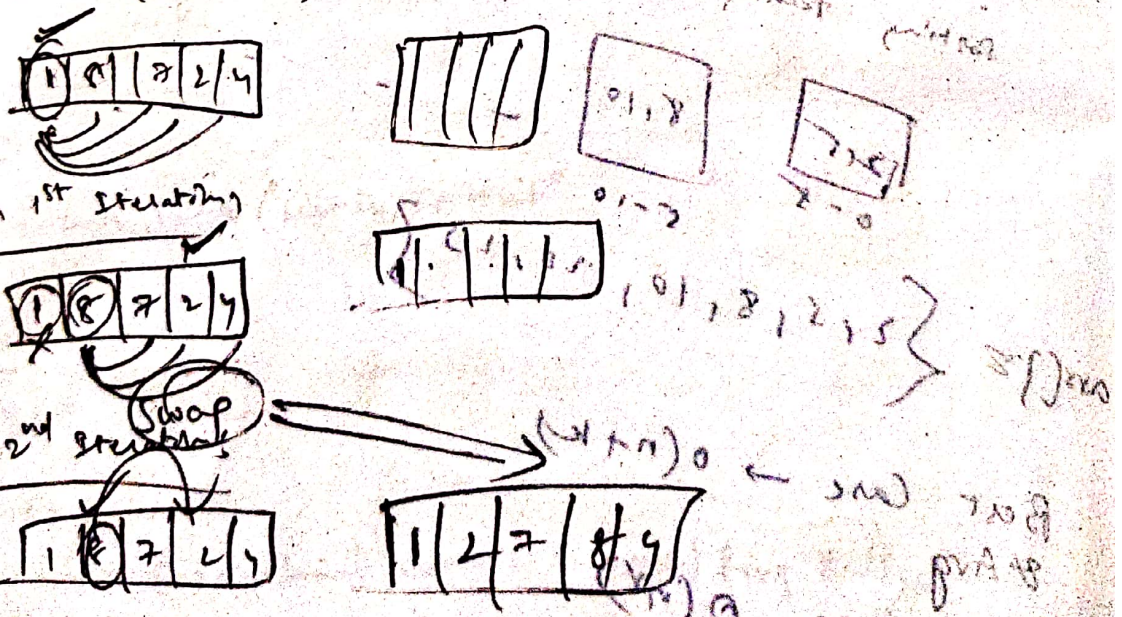
Selection sort: (Time Complexity = O(n²))

One swap per 1 Iteration

At

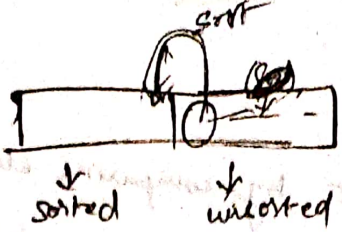
arr = {1, 8, 7, 2, 4}

→ There are two arrays one is sorted another is unsorted (given array)

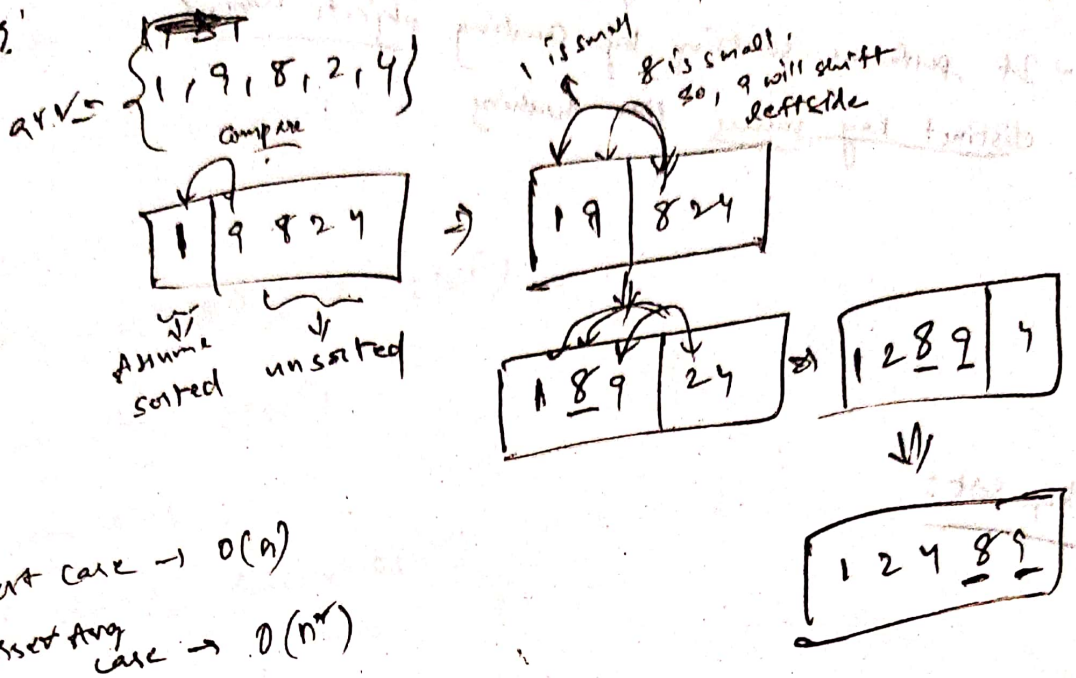


And so on ~

Insertion sort:



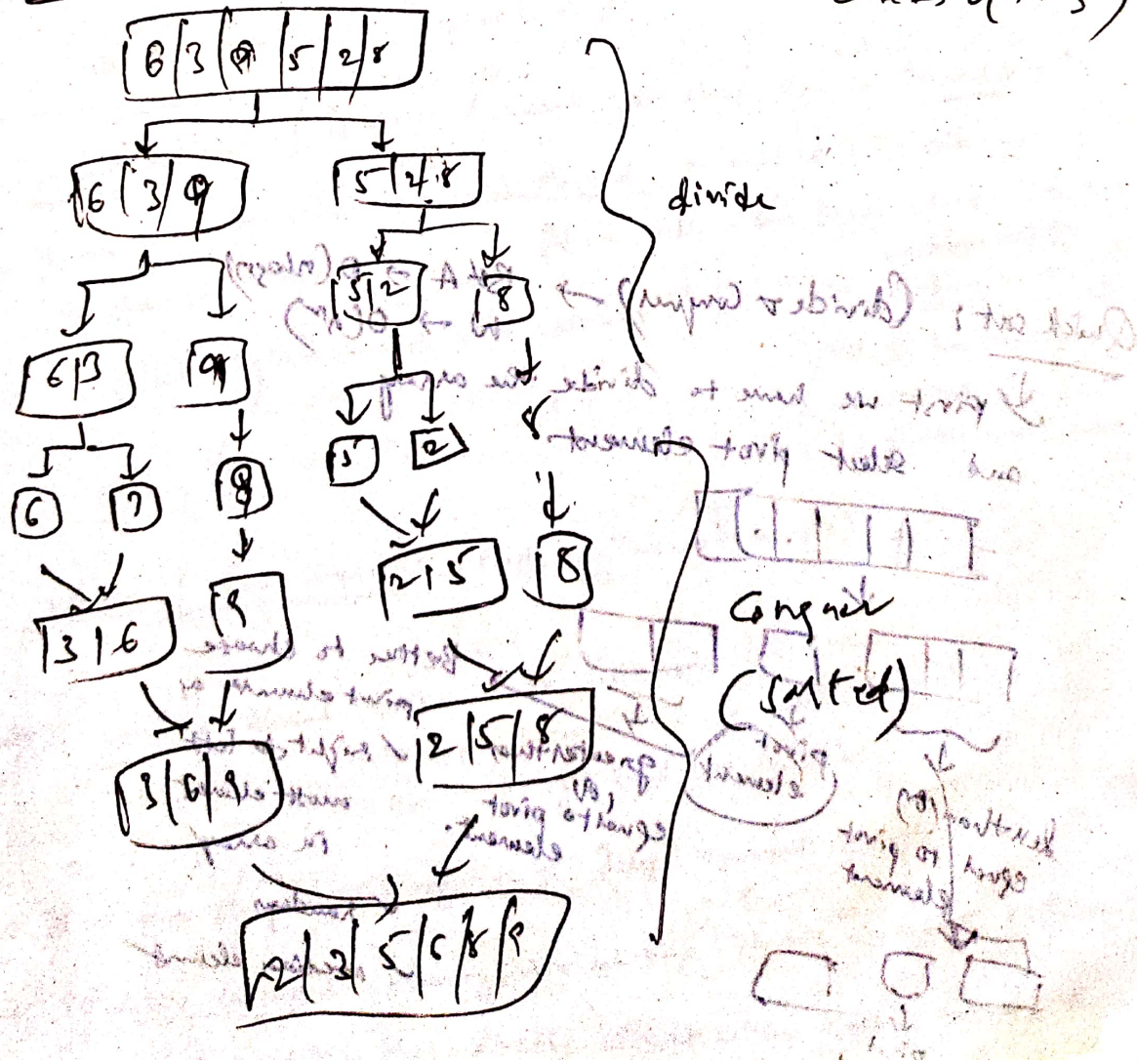
Eg:



✓ Best case → O(n)

Worst & Avg case → O(n²)

Merge sort: (Divide & Conquer) → Best, Avg, Worst case → O(n*logn)



Counting Sort:

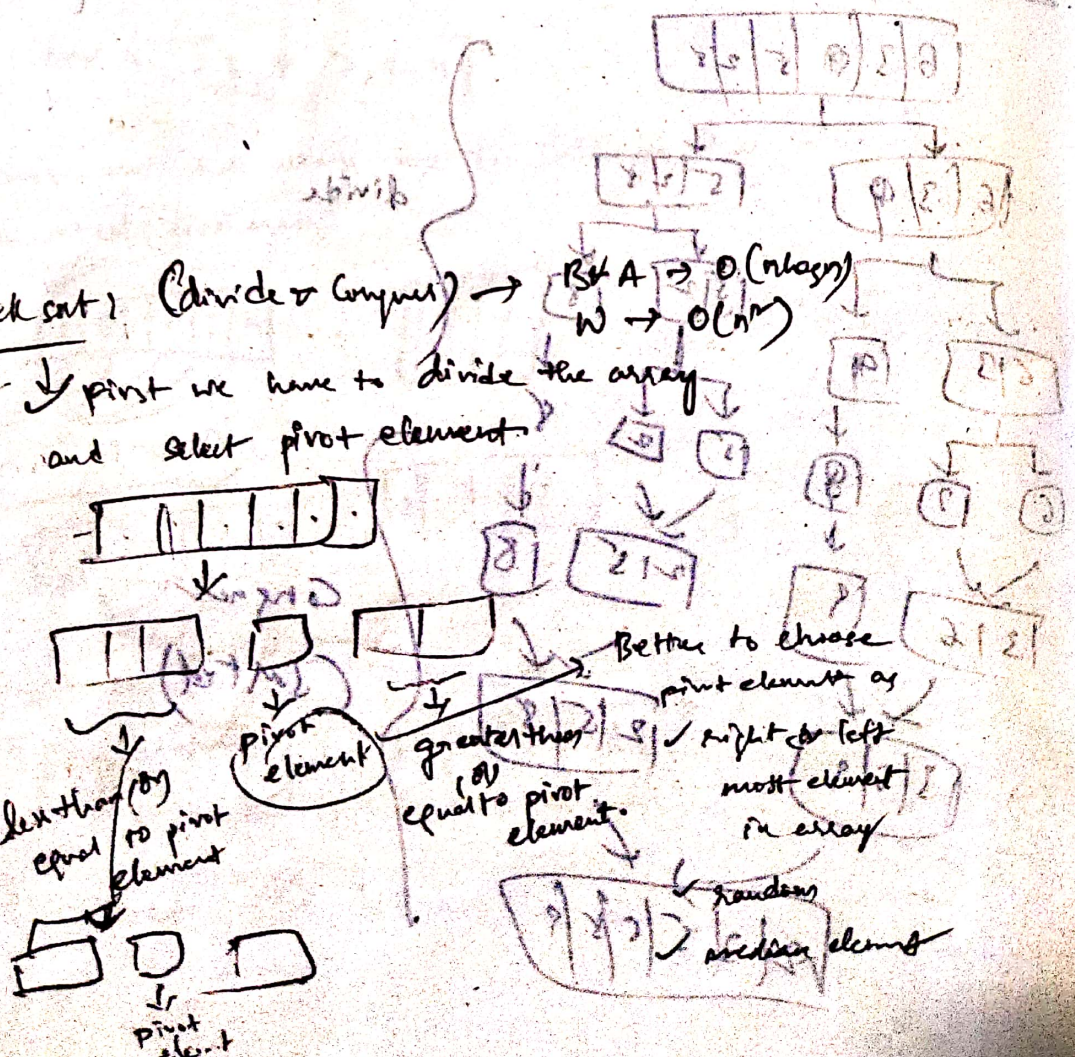
→ It doesn't sort by comparing elements.

→ It performs sorting by counting objects having distinct key values like hashing

Heap Sort:

Quick sort: (Divide & Conquer) → Best A → O(nlogn) Worst → O(n²)

↓ first we have to divide the array and select pivot element.



Bubblesort → ~~B~~

B → O(n), A & W → O(n²)

Bucket sort -

A & B → O(n+k), W → O(n²)

Selection sort = O(n²)

Insertion sort :

B → O(n), A & W → O(n²)

Merge sort :

B, A, W → O(n log n)

Quick sort :

B, A → O(n log n) ; W → O(n²)

Synchronization : (Synchronized means your class/method is thread safe)

+ If we want to work only one thread on method we can use 'Synchronized' keyword.

(If we make method synchronized, then only one thread can work on it)

public synchronized void increment() {

}

(If t1 has working on method, t2 has to wait and viceversa)

Interthread Communication : (Producer-Consumer Example)

→ If we start thread twice → It will run first time → second time it will throw "IllegalThreadStateException".

→ If we give sleep time negative → IllegalArgumentException.

→ Scheduler Criterias : Priority, Arrival time : If both are equal FCFS.

→ If we call run() method directly without calling start() method, the run() method goes onto the current call stack rather than at the beginning of new call stack.

Collections:

Arraylist:

Arraylist <Integer> a = new Arraylist <> ();
(Integer / Class name / String)

→ a.add(5);
a.add(6);
a.add(8);

a.clear(); → Remove all elements

→ Sys(a.lastIndexOf(8)); → Gives index no.

→ Sys(a.contains(8)); → True

→ a.remove(1); → a.get(i) → get the element
(1 → Index no.)

→ a.remove(Integer.valueOf(8)); → Remove value 8

→ Arraylist to Array

```
int arr[] = new int[a.size()];
for (int i=0; i<a.size(); i++)
{ arr[i] = a.get(i);
  Sys(arr[i]);
}
```

→ a.removeIf(n -> (n%8 == 0))

[If Arraylist has different type of elements (int, String) while sorting it will thrown an ClassCastException error.]

→ a.set(2, 8);
(2 → Index no., 8 → value)

→ Collections.sort(a); → ascending
(Collections.sort(a))

→ Collections.reverse(a); → reverse the list

descending

→ Collections.sort(a, Collections.reverseOrder());

First Arraylist sorted in Ascending order, After that by using reverseOrder() method, it will give descending order.

Arraylist Comparator :

```
{ ArrayList<> a = new ArrayList<>();
  a.add(10);  → 5
  a.add(20);  2
  a.add(5);   1
```

→ output 1 2 5 → %10

```
Comparator<Integer> com = new Comp();
→ Collections.sort(a, com);
}
class Comp implements Comparator<Integer> {
    public int Compare(Integer a, Integer b)
                       ↓ classnames
    { if (a%10 > b%10)
        { return 1; → //swap
        }
      return -1; → //not swap
```

Anonymous Inner class

```
class phone
{ show()
  {
  }
main()
```

Inner class :

```
Class A {
int i;
Class B {
}
```

```
A obj = new A();
obj.i = 5;
A.B obj1 = obj.new B();
```

static Inner class :

```
Class A {
static Class B {
    int i;
}
}
```

```
A.B obj = new A.B();
```

Daemon thread provides service to the user threads, when all the user threads dies, JVM terminates this automatically

Multithreading

→ We have to execute the methods parallely when we have more methods. So, to achieve this we have make our class as threads. How to make classes as threads? By extends Thread

If you this, our class before thread start the thread

If we have call start method, we have to override run() method

```
class Hi extends Thread {
    public void run() {
        for(int i=0; i<5; i++)
        { sop(Hi);
          try { Thread.sleep(1000) } catch(Exception e){ } }
    }
    main() { Hi obj1 = new obj1();
             obj1.start() }
```

Criterias: Thread priority, less time to execute

How to implement through Interface:

Functional Interface (It has only run method and it uses lambda expression)

```
class Hi implements Runnable {
    public void run() {
        for(int i=0; i<5; i++)
        { sop(Hi) }
    }
    main()
    { Hi obj1 = new obj1();
      Thread t1 = new Thread(obj1);
      t1.start();
```

Thread t1 = new Thread(obj1) → reference of Interface

Since we didn't have start method in Runnable Interface we can't use it. So, we create start method with thread class

It also accepts name of thread and both combinely

new Thread(obj1, Name_of_thread);

(*) Whenever we want to implement threads, They are 2 methods
→ Extends Thread class
→ Implements Runnable Interface

In Runnable Interface, there is no start() method, so we have create an obj of Thread class and pass the reference obj of Interface.

Characters

public(str.codePointAt(3));
↓
small

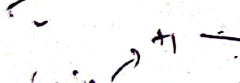
Normalisation: (To avoid redundancy, like insertion, updation and deletion).

1st NF: (Rule: Each attribute of a table must have atomic (single) values)
→ Eliminate multiple values in Column..
→ Create separate column for multiple values.
→ Identify each set of related data with primary key.

2nd NF:
→ Table should be in 1NF.
→ No non-prime attribute is dependent on the proper subset of any (Candidate key of table.

Multithreading: We can use "Lambda Expressions" ~~Runnable Interface~~ Because it is functional Interface.
↳ one thread to wait until the other threads is to finish

join() method: t1.join() → It is a method to make wait thread
t2.join() to join again



When we call join method for t1 & t2 threads, the main thread should have to wait for completion of t1 & t2 execution. After that main thread will execute

isAlive() method: It is boolean. (We can check whether thread is alive (or) not)

t1.getName() → By default Thread-0, Thread-1 ---
↑ return

t1.setName("Hi")

t1.getPriority() → 5 (Range of priority goes from 1 to 10)
(t2.getPriority() → 5
t1.setPriority(1) or t1.setPriority(Thread.MAX_PRIORITY).

Thread Scheduler: It is component in Java that decides which thread to run and which to wait based on Priority & FCFS, Time & arrival.

Collections :

How to Compare two ArrayLists?

ArrayList <Integer> al = new ArrayList <Integer> (Arrays.asList (1,2,3,4,5));

ArrayList <Integer> al2 = new ArrayList <Integer> (Arrays.asList (1,2,4,5,6));

① al.removeAll(al2) → It returns uncommon elements.
sop(al1) // output : 3 (is the uncommon element of many two ArrayLists)

② al1.retainAll(al2) → It returns common elements
sop(al1) // output : [1,2,4,5]

③ public static boolean CompareList (ArrayList al1, ArrayList al2)
{ return al1.toString().contentEquals(al2.toString()) ? true : false;
}
↓ Compare two lists whether the both lists are same or not

④ Arraylist to Array :
String [] item = fruitList.toArray (new String [fruitList.size()]);
↑ list name.

Array to Arraylist :

List <String> item = new ArrayList <> (Arrays.asList(arr));

• How to remove duplicate elements in Arraylist ?
-) Simply assign that Arraylist to 'set' to remove duplicate elements.

Stream API: To process the data in collections.

Once we consumed the stream, we cannot reuse it. [Adv: avoid data leakages]

Stream<Integer> data = nums.stream();

Operations:

List<Integer> nums = Arrays.asList(4,5,6,7,8);

- nums.forEach(n -> System.out.println(n)); // it will print all values

- nums.stream() // creating a stream

- nums.stream().count() // returns size of stream (5)

- nums.stream().map(n -> n*2)

↓

written as)

Stream<Integer> data = nums.stream();

Stream<Integer> mappedData = data.map(n -> n*2);

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

↓ written as-

- nums.stream().map(n -> n*2).forEach(n -> sop(n));

// array = [1,2,3,4,5];

output = [2,4,6,8,10]; -> Stream

I want to multiply only odd numbers

nums.stream().filter(n -> n%2==1).sorted().map(n -> n*2).forEach(n -> sop(n));

apply()

filter() → when we create new Stream, I want to perform operations on only odd numbers.

(It has predicate argument, and predicate is an functional Interface test which returns Boolean value)

nums.stream().filter(n -> n%2==1).sorted().map(n -> n*2).reduce(0,(a,b) -> a+b);

Initial 0+1 0+2=2

→ To convert the stream to Collections like list, set, map etc we use collect.

```
List<String> myList = Arrays.asList("apple", "banana", "cherry");

List<String> collectedList = myList.stream().
                             collect(Collectors.toList());
                                     (or) toSet() (or) toMap()
//result = [apple, banana, cherry];
```

→ Collectors.joining() is to concatenate the elements of a stream into single string.

```
String result = myList.stream().collect(Collectors.joining(","));
//result = apple,banana,cherry.
```

~~Array list to Arrays~~

→ nums.stream().distinct().forEach(sop(n)); // It returns unique elements