

CORE JAVA

Reference : <https://github.com/srihas-sa/CoreJava-DSA.git>

Non-Primitive Data Types

=====]

Non-primitive data types are called reference types because they refer to objects.

The main differences between primitive and non-primitive data types are:

Primitive types in Java are predefined and built into the language, while non-primitive types are created by the programmer (except for String).

Non-primitive types can be used to call methods to perform certain operations, whereas primitive types cannot.

Primitive types start with a lowercase letter (like int), while non-primitive types typically starts with an uppercase letter (like String).

Primitive types always hold a value, whereas non-primitive types can be null.

Examples of non-primitive types are Strings, Arrays, Classes etc.

Java Type Casting

=====

Type casting is when you assign a value of one primitive data type to another type.

In Java, there are two types of casting:

Widening Casting (automatically) - converting a smaller type to a larger type size

byte -> short -> char -> int -> long -> float -> double

Narrowing Casting (manually) - converting a larger type to a smaller size type

double -> float -> long -> int -> char -> short -> byte, ex => double d=20.0 ; int s= (int) d;

SUBCLASS MEANS EXTENDS.

Escape sequences that are valid in Java are:

=====

Code Result Try it

\n New Line

\r Carriage Return

\t Tab

\b Backspace

\f

Math.abs(-2,2) ==> 2 (positive);

int randomNum = (int)(Math.random() * 101); // 0 to 100

Summary of Java Execution Flow

Step Process

1. Write Code Developer writes Java code (.java file).
2. Compilation javac compiles .java → .class (bytecode).
3. Class Loading JVM loads .class into memory.
4. Verification JVM checks for bytecode security & validity.
5. Interpretation & JIT Compilation JVM executes bytecode (interpreted/JIT compiled).
6. Memory Management JVM allocates memory for execution.
7. Execution & Output The program runs and prints output to the console.

METHOD OVERLOADING

=====

SAME CLASSname ,same return type,diff no or type of parameters

blob:<https://web.whatsapp.com/f40cea27-557a-4db5-9f7a-a715e49f70d4>

SUBCLASS MEANS EXTENDS.

ARRAY VS

=====

ARRAY ==> FIXED SIZE, Search entire array for element by index, can store only 1 DATATYPE.

5. When to Use What?

- ✓ Use int when you need better performance and don't need null values.
- ✓ Use Integer when working with collections (List<Integer>, Map<Integer, String>) or when null values are required.

☒ Conclusion:

int is a primitive (faster, no null).

Integer is an object (has methods, can be null, slower).

STRING

=====

String belongs to collection framework.

Memory of strings will be assigned in heap memory

String constant pool where strings will be stored in heap . if there is any common string it won't create new instead points to existing one

String will be immutable

String buffer is mutable and has operations like insert, append, mincapacity

SUBCLASS MEANS EXTENDS.

String builder is similar to buffer but the only diff is related to threads

STATIC

```
class hello {  
  
    static string name;  
    int marks;  
}  
  
h1=new hello();  
hh2=new hello();  
h1.name="srihas";  
print(h1.name,hh2.name) ==> Both will be same as we define in static it is not stored in instance  
variable it will be stored in specific heap for all OBJ it will be same  
static means
```

**Static method

Can use static variables in static function but not non static variables .

Class Loader // static block

```
class details {  
  
    static {  
        string name="SRIHAS";  
    }  
}
```

```
details d1=new d1();
```

SUBCLASS MEANS EXTENDS.

// when the obj is created 1) Class loaded => means static name,static block will get executed
2) objects will be created.

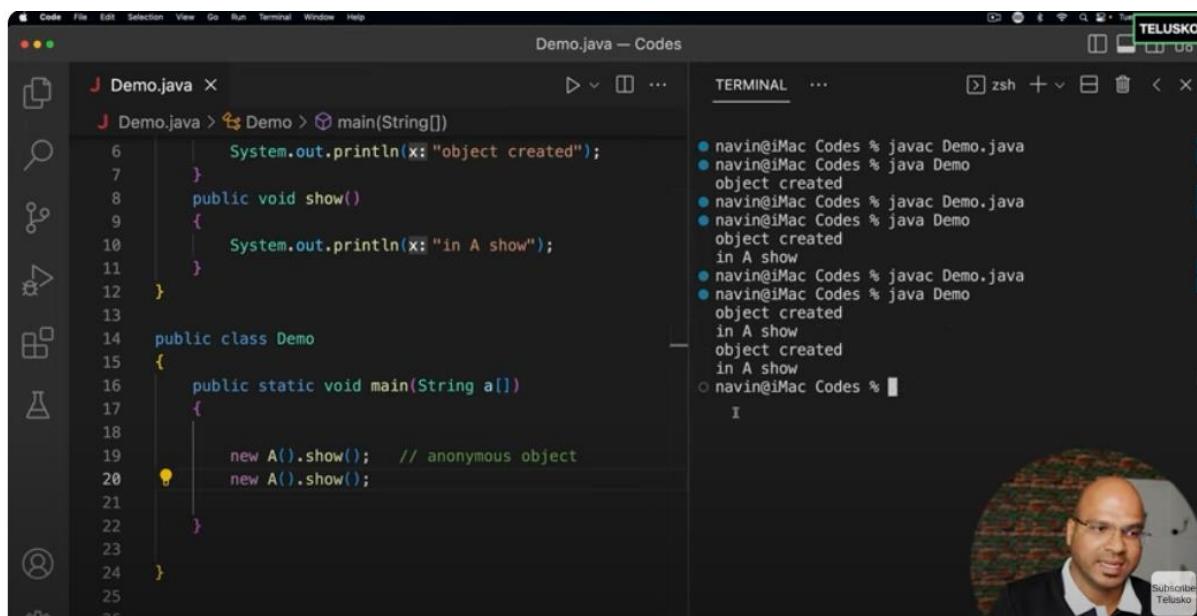
so no matter how many times you create objects the static block will be called once.

Reference vs object creation

Obj a; reference creation.

A=new Obj(); ➔ heap memory creation.

Anonymous object



The screenshot shows a terminal window with the following output:

```
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
object created
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
object created
in A show
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
object created
in A show
object created
in A show
navin@iMac Codes %
```

The terminal window is part of a larger interface, likely a code editor, with tabs for 'TERMINAL' and 'zsh'. There is also a small video player window in the bottom right corner showing a man's face.

Creation of object with out allocation of memory cant use twice the same obj. everytime new obhj will be created

OOPS

Inheritance

```
class Node {  
    int val;  
    Node next;  
    Node prev;  
  
    Node() {}  
  
    public Node(int val, Node next, Node prev) {  
        this.val = val;  
        this.next = next;  
        this.prev = prev;  
    }  
  
    public static void hello1() {  
        System.out.print(s:"hello");  
    }  
}  
  
class AdvHello extends Node {  
    public void advhello() {  
        System.out.println(x:"Adv Hello");  
    }  
}
```

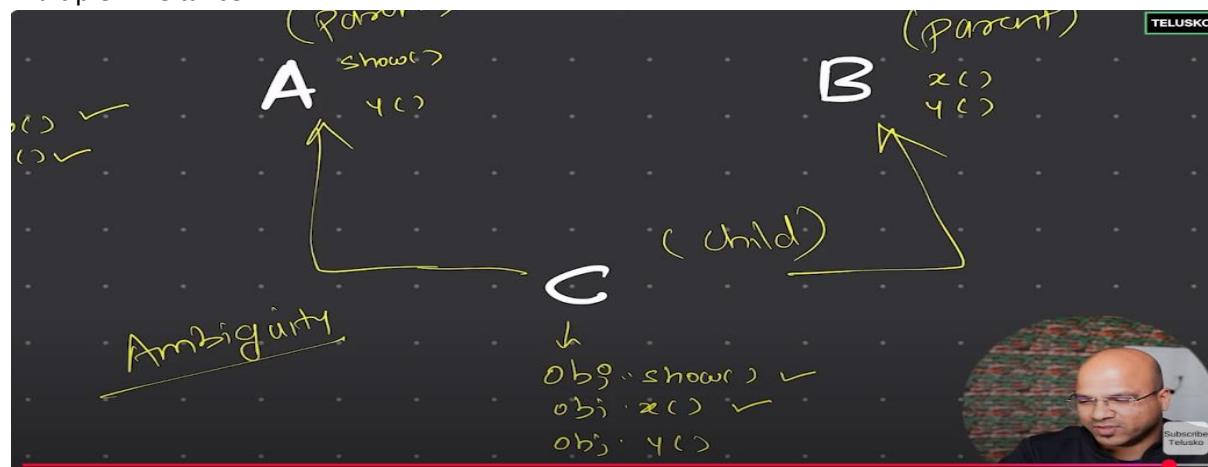
Single level and multilevel inheritance

AdvHello extends Node => **Single Level Inheritance**

If in case there is other class MoreAdv which extends AdvHello

MoreAdv extends AdvHello → AdvHello extends Node → **Multi Level Inheritance**

Multiple Inheritance



Java Not Support Multiple inheritance as it is ambiguous to access y() function present in 2 parents.

SUBCLASS MEANS EXTENDS.

This and Super Keywords

By default every constructor calls super().

Class A {

```
A() {  
    s.o.p("in A");  
}  
  
A(int a){  
    s.o.p("Parametrised Comnstructor for A");  
}  
}
```

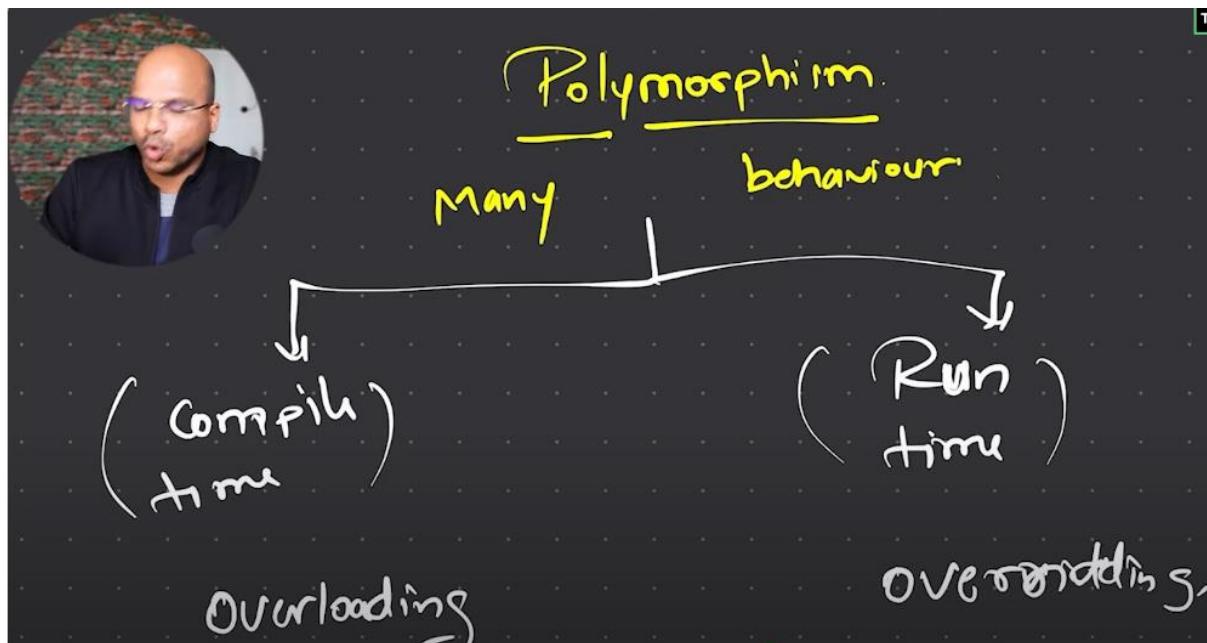
Class B extends A{

```
B() {  
    Super();  
    Sop("in B");  
}  
  
B(int n) {  
    Super();  
    Sop('para in b');  
}  
}
```

B a=new B(); ➔ goes to default con of b then super

Ploymorphism

Many Forms



Method Overloading

Means same method name but different in return types and no of arguments but within same class

Method Overriding

Same Method name and parameters but different class

```
class Calc
{
    public int add(int n1,int n2)
    {
        return n1+n2;
    }
}
class AdvCalc extends Calc
{
    public int add(int n1,int n2)
    {
        return n1+n2+1;
    }
}
```

SUBCLASS MEANS EXTENDS.

```

public class Demo
{
    public static void main(String a[])
    {
        A obj = new A();
        obj.show();

        obj = new B();
        obj.show();

        obj = new C();
        obj.show();
    }
}

```

● navin@iMac Codes %

 ● navin@iMac Codes %

 in A Show

 in B Show

 in C Show

 ○ navin@iMac Codes %

DYNAMIC METHOD DISPATCH OR METHOD OVERRIDING

UNTIL RUN TIME WE DON'T KNOW WHICH SHOW() METHOD IS CALLED EVERY TIME YOU ALLOCATED NEW MEMORY THAT SHOW IS CALLED .

ACCESS MODIFIERS



	Private	Protected	Public	Default
Same class	Yes	Yes	Yes	Yes
Same package subclass	NO	Yes	Yes	Yes
Same package non-subclass	NO	Yes	Yes	Yes
Different package subclass	NO	Yes	Yes	NO
Different package	NO	NO	Yes	NO

PRIVATE → USED WITHIN CLASS

PUBLIC → USED IN SAME AND DIFF PACKAGES

PROTECTED → USED UPTO DIFFERENT PACKAGE BUT ACCESSED IN SUBCLASS MEANS EXTENDING class

SUBCLASS MEANS EXTENDS.

FINAL KEYWORD

VARIABLE ➔ Once defined can't change

CLASS ➔ can't be extended by other class

METHOD ➔ CAN'T BE OVERRIDED

WRAPPER CLASS

Java is not 100% OOPS because it has primitive data types

For collection we need classes not primitive data type so we use wrapper class

Int ➔ Integer so on.... For every primitive data type

```
int a=2;
```

Integer b=a ; ➔ Autoboxing means storing primitive dt to wrapper class

Int c= b; ➔ AutoUnBoxing means changing from class to primitive data type;

ABSTRACT

IF YOU want to declare method but not to initiate then use abstract methos, **also can't create ref to abClass and obj to it only by extending it we can do**

ABSTRACT METHODS SHOUD BE PRESENT ONLY IN ABSTRACT CLASS, NORMAL METHODS OR IMPLEMENTTED METHODS CAN ALSO BE PRESENT.

THE OTHER CLASS THAT EXTENDS ABSTRACT CLASS MUST IMPLEMENT THOSE ALL ABSTRACT METHODS THAT ARE DEFINED IN ABSTRACT CLASS.

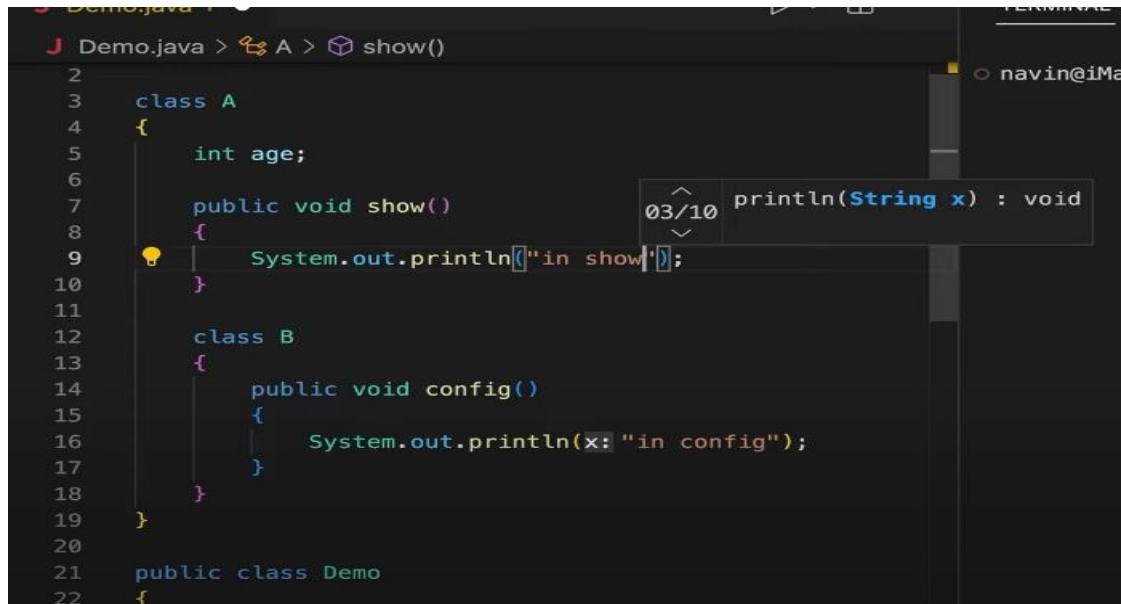
The screenshot shows a Java code editor with the following code:

```
J Demo.java 1 ×
J Demo.java > Car > fly()
1 abstract class Car
2 {
3     public abstract void drive();
4     public abstract void fly();
5
6     public void playMusic()
7     {
8         System.out.println("play music");
9     }
10
11 }
12
13 class WagonR extends Car
14 {
15     public void drive()
16     {
17         System.out.println("Driving..");
18     }
19 }
```

The code defines an abstract class `Car` with two abstract methods: `drive()` and `fly()`. It also contains a concrete method `playMusic()`. A subclass `WagonR` extends `Car` and overrides the `drive()` method. The code is syntax-highlighted with colors for different Java elements.

SUBCLASS MEANS EXTENDS.

INNER CLASS



```
1 Demo.java +  TERMINAL
J Demo.java > A > show()
2
3     class A
4     {
5         int age;
6
7         public void show()
8         {
9             System.out.println("in show");
10        }
11
12         class B
13         {
14             public void config()
15             {
16                 System.out.println("in config");
17             }
18         }
19     }
20
21     public class Demo
22     {
```

For Non Static Inner class we need to use outerclass objects so we create outer class obj then access

```
Outerclass ou = new Outerclass();
```

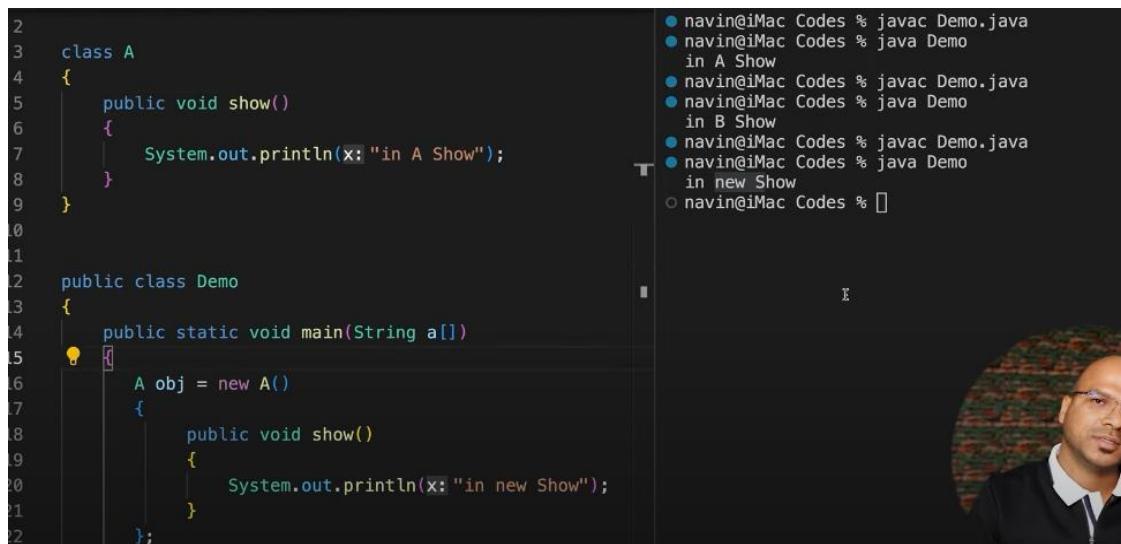
```
Outerclass.Innerclass in = ou.new Innerclass();
```

```
in.show();
```

But For Static Inner class no need to create outer class obj to access as it is a static inner class

```
Outerclass.Innerclass obj=new Outerclass.Innerclass();
```

Anonymous Inner class



```
2
3     class A
4     {
5         public void show()
6         {
7             System.out.println("in A Show");
8         }
9     }
10
11
12     public class Demo
13     {
14         public static void main(String a[])
15         {
16             A obj = new A()
17             {
18                 public void show()
19                 {
20                     System.out.println("in new Show");
21                 }
22             };
23         }
24     }
```

- navin@iMac Codes % javac Demo.java
- navin@iMac Codes % java Demo
- in A Show
- navin@iMac Codes % javac Demo.java
- navin@iMac Codes % java Demo
- in B Show
- navin@iMac Codes % javac Demo.java
- navin@iMac Codes % java Demo
- in new Show
- navin@iMac Codes %

SUBCLASS MEANS EXTENDS.

INTERFACE

INTERFACE ONLY HAVE ABSTRACT METHODS

AND THE CLASS WHICH IMPLEMENTS INTERFACE SHOULD CALL ALL METHODS

```
// class - class -> extends  
// class - interface -> implements  
// interface - interface -> extends|
```

```
package Interface;
```

```
interface Common {
```

```
    void hello();
```

```
}
```

```
class Inter implements Common {
```

```
    public int i = 5;
```

```
    public void hello() {
```

```
        System.out.print("In Inter");
```

```
}
```

```
}
```

```
class Inter1 implements Common {
```

```
    public void hello() {
```

```
        System.out.print("In Inter1");
```

```
}
```

```
}
```

```
class ii {
```

```
    public void pp(Common obj) {
```

```
        obj.hello();
```

SUBCLASS MEANS EXTENDS.

```

    }
}

public class Interf {
    public static void main(String[] args) {
        Common a = new Inter();
        Common bs = new Inter1();
        ii i = new ii();
        i.pp(a);
        // prints 5
    }
}

```

CAN IMPLEMENT TWO INTERFACES

```

interface Common2 {
    void hello1();
}

interface Common1 extends Common {
    void hello1();
}

class Hello implements Common1, Common2 {
    public void hello() {
        System.out.print("Hello");
    }

    public void hello1() {
        System.out.print("Hello World");
    }
}

class Inter implements Common {
    public int i = 5;
}

```

HERE unlike we cant extend 2 classes so multiple inheritance fails

- **But in Interfaces we can implement 2 interfaces as they don't have any definition there won't be ambiguity problem in interfaces**

SUBCLASS MEANS EXTENDS.

ENUM

The screenshot shows a Java code editor with a file named `Demo.java`. The code defines an enum `Status` with values `Running`, `Failed`, `Pending`, and `Success`. It also contains a `main` method that prints the ordinal value of the `Status.Success` constant. To the right, a terminal window shows the output of running the code twice, both times producing the output `0`.

```
Demo.java 1 X
Demo.java > Demo > main(String[])
1
2     enum Status{
3         Running, Failed, Pending, Success;
4     }
5
6     public class Demo
7     {
8         public static void main(String a[])
9         {
10             int i = 5;
11             Status s = Status.Success;
12             System.out.println(s.ordinal());
13         }
14     }
TERMINAL ... zsh + ...
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Running
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
Failed
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
0
navin@iMac Codes % javac Demo.java
navin@iMac Codes % java Demo
3
navin@iMac Codes %
```

Enum means it is also a class inside there will be named constants instead of using string or char.

The screenshot shows a Java code editor with a file named `Enum.java`. The code defines a class `Enum` with a `main` method. Inside `main`, it creates an enum `Laptop` with three constants: `Ideapad`, `macbook`, and `Lenovo`, each associated with a price. It then creates an instance `e1` of `Laptop.Ideapad` and prints its price. The output of running the code is `2000`.

```
public class Enum {
Run | Debug
    public static void main(String args[]) {
        enum Laptop {
            Ideapad(price:2000), macbook(price:100), Lenovo(price:3000);

            int price;

            Laptop(int price) {
                this.price = price;
            }

        }
        ;
        Laptop e1 = Laptop.Ideapad;
        System.out.print(e1.price);
    }
}
```

SUBCLASS MEANS EXTENDS.

Annotations

Annotation means interacting with compiler and find errors during compile time itself. Gives red line

@override ➔ Used to override the method

@ FunctionalInterface ➔ Restrict to have only one method inside interface.

Lambda Expressions

Should be used with @ FunctionalInterface

```
package LambdaExp;
```

```
interface A {  
    public void show();  
}
```

```
interface B {  
    public void show(int i);  
}
```

```
public class Lambda {  
    public static void main(String args[]) {  
        // 1st--> Way of defining Lambda Expressions  
        A obj = new A() {  
            public void show() {  
                System.out.println("Hello Lambda");  
            }  
        };  
        obj.show();  
        // 2nd--> Way of defining Lambda Expressions  
        A obj1 = () -> System.out.println("Hello in Lambda2");  
        obj1.show();  
    }  
}
```

```

// Lambda expr with parameters

B obj2 = new B() {
    public void show(int i) {
        System.out.println("Hello Lambda" + i);
    }
};

obj2.show(5);

B obj3 = (i) -> System.out.println("Hello in Lambda2" + i);
obj3.show(10);

}

}

```

Types of Interfaces

Normal Interfaces ➔ Having multiple methods

Functional Inter or SAM ➔ Single abstract Method

Marker Interface ➔ No Methods used for serializing and deserializing

Exception

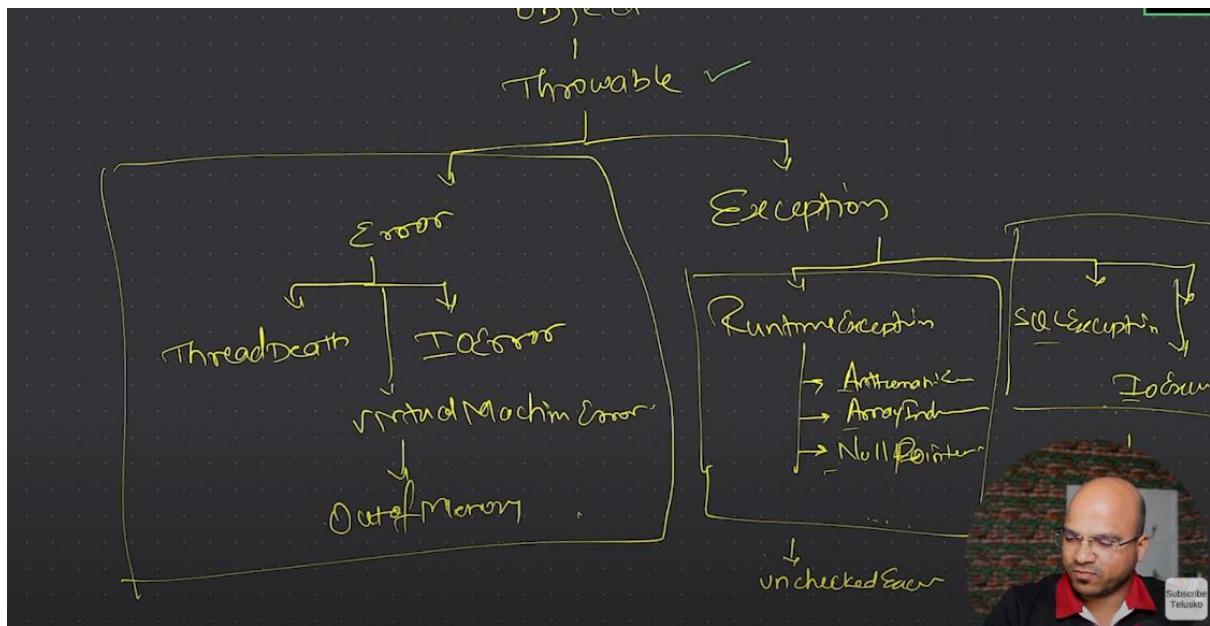


```

try
{
    j = 18/i;
    System.out.println(nums[1]);
    System.out.println(nums[5]);
}
catch(ArithmeticException e)
{
    System.out.println(x: "Cannot divide by zero" );
}
catch(ArrayIndexOutOfBoundsException e)
{
    System.out.println(x: "Stay in your limit.");
}
catch(Exception e)
{
    System.out.println(x: "Something Went w[]");
}

```

SUBCLASS MEANS EXTENDS.



In Exception we have checked and unchecked

Unchecked Exception → Means not force you to handle the exception.

Checked Exception → Means You should Handle the exception like sql exception ,etc

Errors → Something you can't handle .

For More Reference go through the git hub repository

Link: <https://github.com/srihas-sa/CoreJava-DSA.git>

Input/Output

```

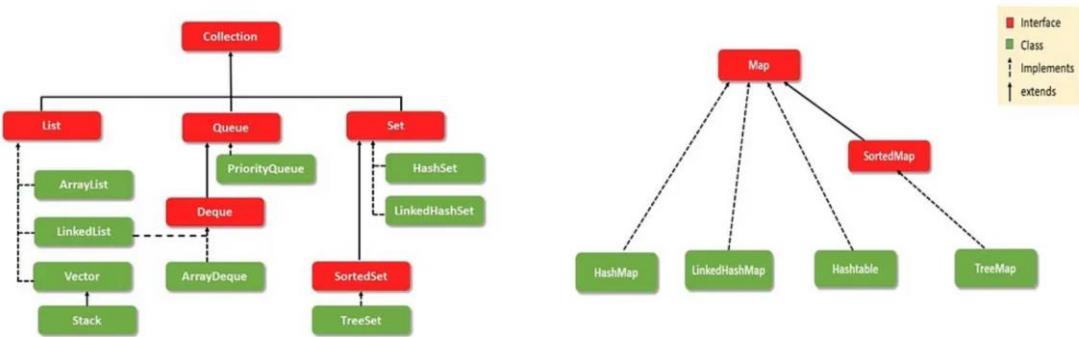
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

public class Inou {
    Run | Debug
    public static void main(String[] args) throws NumberFormatException, IOException {
        // 1st Way of taking input older versions
        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(in);
        int n = Integer.parseInt(br.readLine());
        System.out.print(n);
        // 2nd way
        Scanner sc = new Scanner(System.in);
        int n1 = sc.nextInt();
        sc.nextLine();
        String s = sc.nextLine();
        System.out.print(s);
    }
}

```

SUBCLASS MEANS EXTENDS.

COLLECTION FRAMEWORK IN JAVA



Collection is a interface which have methods like add etc which will be implemented by all classes

Collection extends by other interfaces which provide Specific Operation usefull for that Interface

****That extended Interface will be implemented by different classes no need to define or create anonymous inner class for the interface we have build in classes which will implement those methods or operations inside that particular interface . that's why we call it Collection API

Set

Hashset → Unique ,UnSorted

Treeset→Unique,Sorted.

Map

HashMap→Normal

HashTable→Synchronous like threads

We can use Collections to sort



```
J Demo.java 3 ● J Comparator.class J TreeSet.cl... ▶ ⓘ ... TERMINAL ... zsh + ⓘ
J Demo.java > Demo > main(String[])
25     public static void main(String a[])
26     {
27         Comparator<Integer> com = new Comparator<Integer>()
28         {
29             public int compare(Integer i, Integer j)
30             {
31                 if(i%10 > j%10)
32                     return 1;
33                 else
34                     return -1;
35             }
36         };
37
38         List<Student> nums = new ArrayList<>();
39         nums.add(new Student(age: 21, name: "Navin"));
40         nums.add(new Student(age: 12, name: "John"));
41         nums.add(new Student(age: 18, name: "Parul"));
42         nums.add(new Student[], "Navin"));
43
44         Collections.sort(nums, com);
45
```

We can use Collections to sort by default it will sort

```
Collections.sort(nums);
```

But if you want to implement your own logic like compare unit's place and sort we need to implement that method called Comparator which will be used to Sort based on our logic

*****STACK AND HEAP MEMORY MANAGEMENT AND GARBAGE COLLECTION(GC)**

```
Void main{  
    Int a=2; //Stores in Stack memory in main method scope  
    Class a = new Class; // Ref will be stored in main scope but address in heap memory  
    a.call() // calls and from here on New Scope will be created in stack.  
}
```

```
Class a{  
    Public void call() // Stores in scope of call method  
}
```

Once execution of scope is done like after closing Bracket stack will be freed in Last in First Out Manner Even references will be deleted.

So first call method scope will be deleted

Then Main method and it's Referencees

Then in Heap Memory Not referencing Objects will be deleted Automatically by **Garbage collector**

System.gc(); // JVM decide when it runs even though you executed command

Based on space it will be running accordingly.

Strong Reference ➔ How we create Class a = new Class; can't be deleted until ref cleared by garbage coll

Weak Reference ➔ WeekReference<Person> p=new ... ➔ As soon as GC runs it will be cleared

Soft Reference ➔ Similar but removed only if it is urgent like no memory .

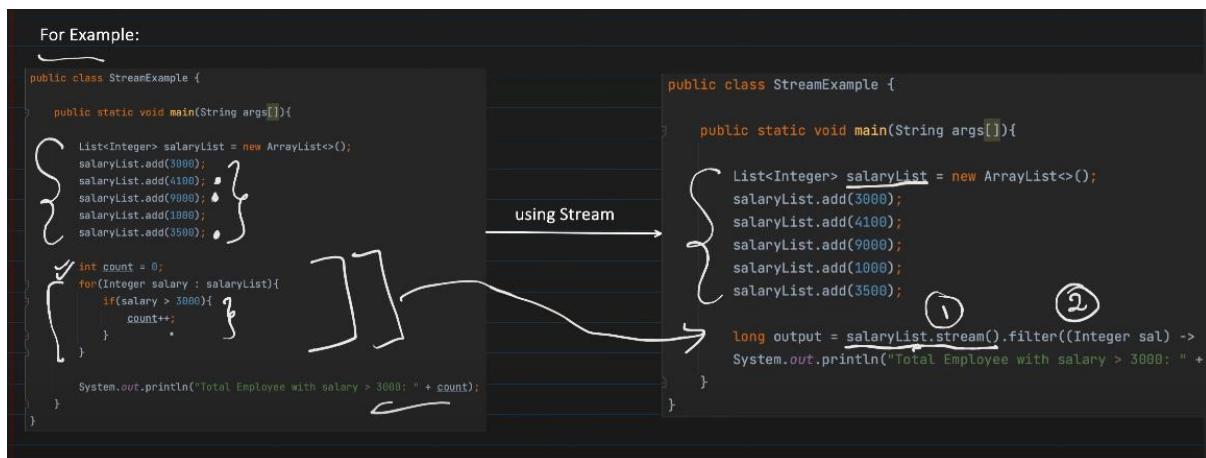
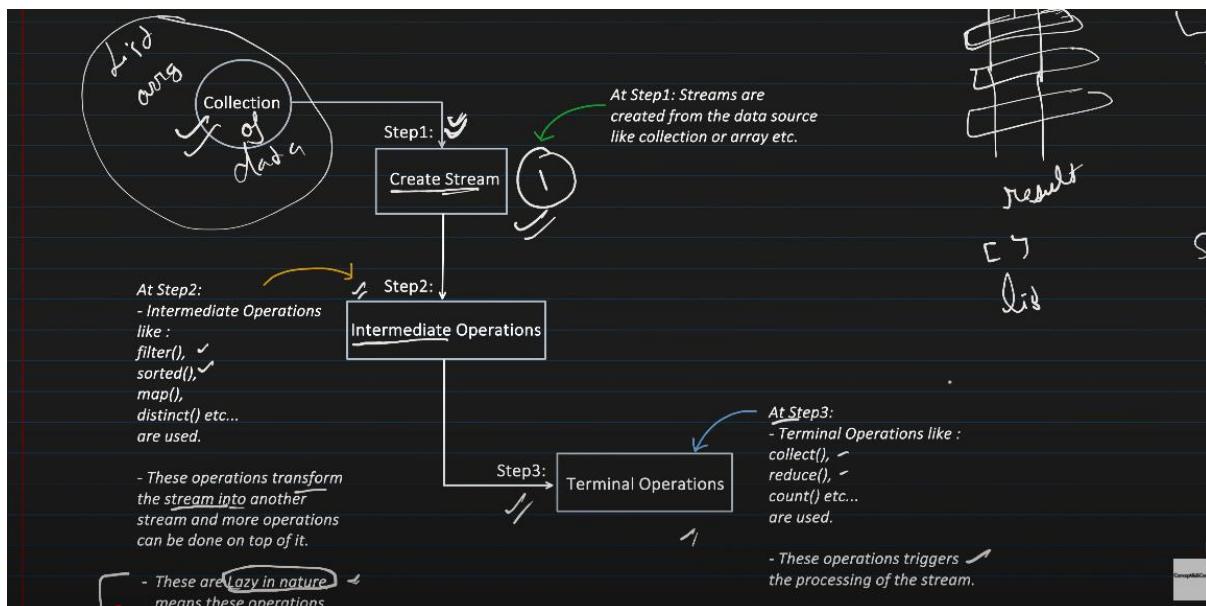
Heap memory space Young age,old generation and metadata

Means if obj is present in longer time it is stored in old or else in new

Mark and Sweep algo, in young gen gc will run more freq than old gen

Meta data ➔ All static data,class data will be stored,also called class loader

STREAMS



Only be executed once the terminal operation is called

Filter, map, these all are intermediate operations they are called lazy because they wont be executed without terminal operations like count, toArray() etc ...

Sequence of Stream Operations:

Expected Output:	Actual Output:
after filter: 4	after filter:4
after filter: 7	after negating:-4
after filter: 10	after filter:7
after negating: -4	after negating:-7
after negating: -7	after filter:10
after negating: -10	after negating:-1
after Sorted: -10	after Sorted: -7
after Sorted: -7	after Sorted: -7
after Sorted: -4	after Sorted: -4

SUBCLASS MEANS EXTENDS.

Parallel Stream:

Helps to perform operation on stream concurrently, taking advantage of multi core CPU.
ParallelStream() method is used instead of regular stream() method.

Internally it does:

- Task splitting : it uses "spliterator" function to split the data into multiple chunks.
- Task submission and parallel processing : Uses Fork-Join pool technique.

Note: i will cover Fork-Join pool implementation in Multithreading topic

Work faster than Normal Stream because it will divide task into sub tasks.

GENERIC CLASSES

```
public class Print<T> {  
    T value;  
    public T getPrintValue(){  
        return value;  
    }  
    public void setPrintValue(T value){  
        this.value = value;  
    }  
}
```

Print<Integer> ii=new Print<>();

SUBCLASS MEANS EXTENDS.

****SINGLETON CLASS****

Have Single class and Single Object

For Db Connection we just need one instance don't want multiple obj so then we use these ,.

Singleton Class: ~~Immutable Class~~

This class objective is to create only 1 and 1 Object.

Different Ways of creating Singleton Class:

- Eager Initialization
- Lazy Initialization
- Synchronization Block
- Double Check Lock (there is a memory issue, resolved through Volatile instance variable)
- Bill Pugh Solution
- Enum Singleton

Eager Initialization

Here we create static method to create new object and make constructor as private so we can't create object with new keyword

During loading itself it will load the static methods and create object whether you use it or not that is disadvantage.

```
public class DBConnection {  
    ① private static DBConnection conObject = new DBConnection();  
    ② private DBConnection(){  
    }  
    ③ public static DBConnection getInstance(){  
        return conObject;  
    }  
}  
  
public class Main {  
    public static void main(String args[]){  
        DBConnection connObject = DBConnection.getInstance();  
    }  
}
```

SUBCLASS MEANS EXTENDS.

Lazy Initialization

Object is Created Not in static block in main method if it is Not Null then New Obj will be created

```
(1) private static DBConnection conObject; = null  
(2) private DBConnection(){  
}  
  
(3) public static DBConnection getInstance(){  
    if(conObject == null){  
        conObject = new DBConnection();  
    }  
    return conObject;  
}
```

Advantages

Object is not created Before hand

Disadvantages

If Multiple Thread come at a time and if obj is not already Created both will create New Objects();

Synchronization Block

Same as above but Just as Synchronized Before the method getInstance() so only one thread at a time can use that function

Lock and Unlock Method so no multiple threads can use

Advantages

Will not create more than one object

Disadvantages

Too Slow because of locking and unlocking.

Double Locking:

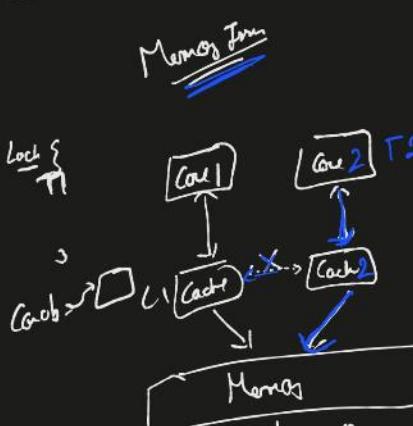
```
public class DatabaseConnection {  
    private static volatile DatabaseConnection conObject; // ①  
    private DatabaseConnection() { // ②  
    }  
  
    public static DatabaseConnection getInstance() { // ③  
        if(conObject == null){ // T1  
            synchronized (DatabaseConnection.class){ // Lock {  
                if(conObject == null){ // T2  
                    conObject = new DatabaseConnection();  
                }  
            }  
        }  
        return conObject;  
    }  
}
```

Here We use Double check even if two threads enters to method 1st only one will go to inside creation.

Issue with Memory

Here in above without volatile keyword the memory allocation will happen in catch not in main memory so it will create 2 obj in two catch so to avoid catch and directly work with main memory we use VOLATILE

```
public class DatabaseConnection {  
    private static volatile DatabaseConnection conObject; // ①  
    private DatabaseConnection() { // ②  
    }  
  
    public static DatabaseConnection getInstance() { // ③  
        if(conObject == null){ // T1  
            synchronized (DatabaseConnection.class){ // Lock {  
                if(conObject == null){ // T2  
                    conObject = new DatabaseConnection();  
                }  
            }  
        }  
        return conObject;  
    }  
}
```



Bill Plug Solution

The code shows the implementation of a Singleton pattern:

```
public class DatabaseConnection {
    private DatabaseConnection() {
        // ①
    }

    private static class DBConnectionHelper {
        // ②
        private static final DatabaseConnection INSTANCE_OBJECT = new DatabaseConnection();
    }

    // ③
    public static DatabaseConnection getInstance() {
        return DBConnectionHelper.INSTANCE_OBJECT;
    }
}
```

Annotations:

- ①: A circled '1' above the constructor.
- ②: A circled '2' above the static inner class declaration.
- ③: A circled '3' above the static method declaration.
- A handwritten arrow points from the circled '2' to the circled '3', indicating the relationship between the static helper class and the static method.

Here we use **static inner class**

Whenever Getinstance() is called for the 1st time static inner class will be executed and create new object

As it is static inner class it will load directly without creating again.

ENUM

The code shows the implementation of a Singleton pattern using an Enum:

```
enum DBConnection {
    INSTANCE;
}
```

Annotations:

- ENUM: A circled 'ENUM:' with a checkmark.
- JVM: A handwritten note 'JVM' with a checkmark.
- private: A handwritten note 'private' with a checkmark.

By default Enum has Private Constructor and only one instance will be present

So Enum is Singleton Class.

SUBCLASS MEANS EXTENDS.

Immutable Classes can't be changed and modified and inherited

IMMUTABLE CLASS: ✓
NOT ✗
Document ✗

- We can not change the value of an object once it is created.
- Declare class as 'final' so that it can not be extended.
- All class members should be private. So that direct access can be avoided.
- And class members are initialized only once using constructor.
- There should not be any setter methods, which is generally used to change the value.
- Just getter methods. And returns Copy of the member variable.
- Example: String, Wrapper Classes etc.

```
final class MyImmutableClass {  
    private final String name;  
    private final List<Object> petNameList;  
  
    MyImmutableClass(String name, List<Object> petNameList){  
        this.name = name;  
        this.petNameList = petNameList;  
    }  
  
    public String getName(){  
        return name;  
    }  
}
```

No Setter ,Final class,Final Variables

Java8 Features Default Method in Interfaces

abstract ✓
✓ **unplanned** ✓

```
public interface Bird {  
    ✓ public void canFly();  
  
    default int getMinimumFlyHeight(){  
        return 100;  
    }  
}  
  
public class Eagle implements Bird{  
  
    @Override  
    public void canFly() {  
        //eagle fly implementation  
    }  
}  
  
public class Sparrow implements Bird {  
  
    @Override  
    public void canFly() {  
        //sparrow fly logic  
    }  
}
```

Now we can provide **implementation to the Method in Interface using default Keyword.**

SUBCLASS MEANS EXTENDS.

Interface

3. Private Method and Private Static method (Java9):

- ✓ We can provide the implementation of method but as a private access modifier in interface.
- ✓ It brings more readability of the code. For example if multiple default method share some code, t
- ✓ It can be defined as static and non-static.
 - From Static method, we can call only private static interface method.
 - Private static method, can be called from both static and non static method.
 - Private interface method can not be abstract. Means we have to provide the definition.
 - It can be used inside of the particular interface only.

```
public interface Bird {  
    void canFly(); //this is equivalent to public abstract void canFly();  
  
    public default void minimumFlyingHeight() {  
        myStaticPublicMethod(); //calling static method  
        myPrivateMethod(); //calling private method  
        myPrivateStaticMethod(); //calling private static method  
    }  
  
    static void myStaticPublicMethod() {  
        myPrivateStaticMethod(); //from static we can call other static method  
    }  
  
    private void myPrivateMethod(){  
        // private method implementation  
    }  
  
    private static void myPrivateStaticMethod(){  
        // private static method implementation  
    }  
}
```

Bird

private static

Functional Interfaces

```
@FunctionalInterface  
public interface Bind {  
  
    void canFly(String val); } Abstract Method  
  
    default void getHeight(){ //default method implementation } Default Method  
  
    static void canEat(){ //my static method implementation } Static Method  
  
    String toString(); //Object class method
```

*** Functional Interfaces should have one abstract method and n number of default and static methods as they will have implementation they will no longer be abstract methods.

Types of Functional Interfaces

```
@FunctionalInterface  
public interface Consumer<T> {  
    void accept(T t); } Consumer  
  
@FunctionalInterface  
public interface Supplier<T> {  
    T get(); } Supplier
```

Left one is Consumer

```
public class Main {  
    public static void main(String args[]) {  
        Consumer<Integer> loggingObject = (Integer val) → {  
            if(val>10) {  
                System.out.println("Logging");  
            }  
        };  
        loggingObject.accept( t: 11);  
    }  
}
```

Right One is Supplier => get from method

```
public class Main {  
    public static void main(String args[]) {  
        Supplier<String> isEvenNumber = () -> "this is even";  
        System.out.println(isEvenNumber.get());  
    }  
}
```

OR

1) Consumer ➔ Accepts Value Not return

2) Supplier ➔ Returns Value

3) Function ➔ Accepts and return value

4) Predicate ➔ Return Boolean

Reflection Class

1. What is Reflection?

This is used to examine the Classes, Methods, Fields, Interfaces at runtime and also possible to change the behavior of the Class too.

For example:

- o What all methods present in the class.
- o What all fields present in the class.
- o What is the return type of the method.
- o What is the Modifier of the Class
- o What all interfaces class has implemented
- o Change the value of the public and private fields of the Class etc.....

```
class Bird { }

//get the object of Class for getting the metadata information
Class birdClass = Class.forName( className: "Bird");
```

2. Using .class

```
//assume that we have one class called Bird
class Bird { }

//get the object of Class for getting the metadata information of
Class birdClass = Bird.class;
```

3. Using getClass() method

```
//assume that we have one class called Bird
class Bird { }

Bird birdObj = new Bird();
```

Reflection Classes Shows the methods ,parameters,fields thhta are present in the class

We can create the Reflecion class by using Class Object.

Internally for Every Class one Class obj will be created we can access by above methods.

```
public class Eagle {
    Eagle() {
    }

    public void fly(int paramInt, boolean boolParam, String strParam) {
        System.out.println("fly paramInt: " + paramInt + " boolParam: " + boolParam + " strParam: " + strParam);
    }
}

public class Main {
    public static void main(String args[]) throws InstantiationException, IllegalAccessException, ClassNotFoundException {
        Class eagleClass = Class.forName( className: "Eagle");
        Object eagleObject = eagleClass.newInstance();

        Method flyMethod = eagleClass.getMethod( name: "fly", ...parameterTypes: int.class, boolean.class, String.class);
        flyMethod.invoke(eagleObject, ...args: 1, true, "Hello");
    }
}
```

By using reflector class you can even change and access private methods and variables

SO you don't have access specifications in relection

That's why singleton CLASS ALSO NOT APPLICATE AS ITS CONSTRUCTOR IS PRIVATE WE CAN CALL IT BY CREATING CLASS AND CHANGE ACCESS SPECIFICATION AND CALL THAT CONSTRUCTOR WHICH IS PRIVATE.

NOT RECOMMEND USING IT

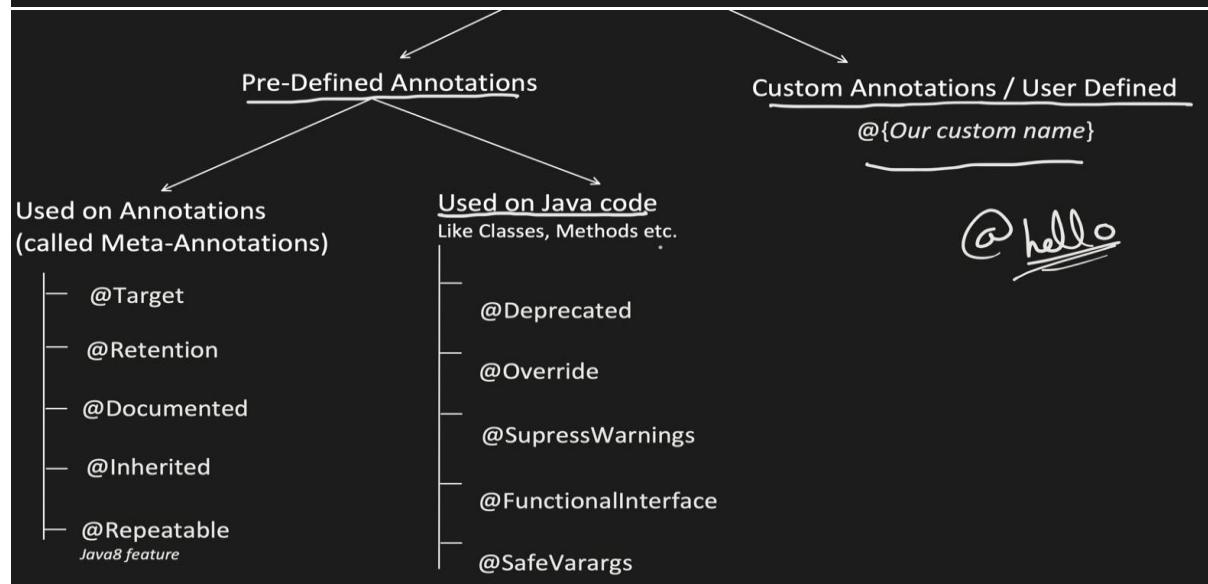
BREAK SINGLETON BY BREAKING PRIVATE

SUBCLASS MEANS EXTENDS.

ANNOTATION

What is Annotation?

- ✓ It is kind of adding **META DATA** to the java code.
- ✗ Means, its usage is **OPTIONAL**.
- ✓ We can use this meta data information at runtime and can add certain logic in our code if wanted.
- ✓ How to Read Meta data information? Using **Reflection** as discussed in previous video.
 - Annotations can be applied at anywhere like Classes, Methods, Interface, fields, parameters etc.



@Deprecated : Shows Warning when ever you used that particular method,filed,class etc..

@SupressWarning : Do not give warnings including depreciated warnings

Meta Annotation

Annotation over Annotation

Like

@Target

@Override

Target=@ ElementType.Method ➔ means used only on method

@Retention (RetentionPolicy.**Source**) ➔ Present only in **Source code** not in class file

(RetentionPolicy.**Class**) ➔ Present **Class file** but ignored by jvm at **RunTime**.

@Retention (RetentionPolicy.**RunTime**) ➔ During Run time also it will be present.

@Documented ➔ If you want to show annotation in java Document use it.

@Inherited ➔ Child class also can able to use the parent class Annotations.

SUBCLASS MEANS EXTENDS.

We need to use @Repeable Meta-annotation

```

@Repeatable(Categories.class)
@interface Category {
    String name();
}

@Retention(RetentionPolicy.RUNTIME)
@interface Categories {
    Category[] value();
}

```

```

public class Eagle{
    public void fly() {
    }
}

```

Creating an Annotation with method (its more like a field):

- No parameter, no body.
- Return type is restricted to Primitive, Class, String, enums, annotations and array of these types.

```

public @interface MyCustomAnnotation {
    String name();
}

```

```

@MyCustomAnnotation(name = "testing")
public class Eagle{
    public void fly() {
    }
}

```

NO	METHODS	Available in	USAGE
1.	✓ size()	Java1.2+	✓ It returns the total number of elements present in the collection.
2.	✓ isEmpty()	Java1.2	Used to check if collection is empty or has some value. It return true/false.
3.	✓ contains()	Java1.2	Used to search an element in the collection, returns true/false.
4.	✓ toArray()	Java1.2	It convert collection into an Array.
5.	✓ add()	Java1.2	Used to insert an element in the collection.
6.	✓ remove()	Java1.2	Used to remove an element from the collection.
7.	✓ addAll()	Java1.2	Used to insert one collection in another collection.
8.	✓ removeAll()	Java1.2	Remove all the elements from the collections, which are present in the collection passed in the parameter.
9.	✓ clear()	Java1.2	Remove all the elements from the collection.
10.	✓ equals()	Java1.2	Used to check if 2 collections are equal or not.
	✓ stream()		

SUBCLASS MEANS EXTENDS.

COLLECTIONS

QUEUE → DEQUEUE → Double Sided Queue

```
package Collections;

import java.util.ArrayDeque;
import java.util.Collection;
import java.util.Deque;
import java.util.Queue;
import java.util.concurrent.DelayQueue;

public class Dequeue {
    Run | Debug
    public static void main(String[] args) {
        Deque<Integer> cl = new ArrayDeque<>();
        cl.add(1);
        cl.addFirst(0);
        cl.addLast(2);
        cl.removeFirst();
        cl.removeLast();
        cl.getFirst();
        cl.getLast();
        System.out.println(cl);
    }
}
```

Insertion T.C → O(1) Mostly except when size got full it copies to new deque with double size then it will be O(n);

Deletion → O(1)

Search → O(1)

List

CAN BE INSERTED AND DELETED IN MIDDLE IT IS INDEX BASED UNLIKE QUEUE .

INTERNAL USES ARRAY DATA STRUCTURE

```
ListIterator<Integer> li = cl1.listIterator();
while (li.hasNext()) {
    System.out.println(li.next());
}

while (li.hasPrevious()) {
    System.out.println(li.previous());
}

// if you want to write your own logic for sorting then
System.out.println(cl1);
```

SUBCLASS MEANS EXTENDS.

Can travel backward through ListIterator()

LinkedList

Implement Both Queue and List

Can addfirst,last,search,middle,set,get

Insertion T.C \rightarrow O(1);

Search \rightarrow O(n);

VECTOR

Same As ArrayList

But ThreadSafe

Have locks

*****HASHMAP*****

Internal Implementation of Hashmap

Node { hash:,key:,val:,next:}

Internally there will be ARRAY [16]

Map.put(1,20) \rightarrow 1 will go to hasing \rightarrow 12002(hashValue) \rightarrow HashVal%LenArray \rightarrow Store in that index

If you get Same val with other key then it will go to the same index but now it already have value so it will store in .next (LINKED LIST) .

Internally it uses load Factor if all are filled then 2x the storage and refactor again

If there were More than theroshold val linked in a linked list then

Linked List will be converted to Balanced Binary tree so from O(n) to O(logn)

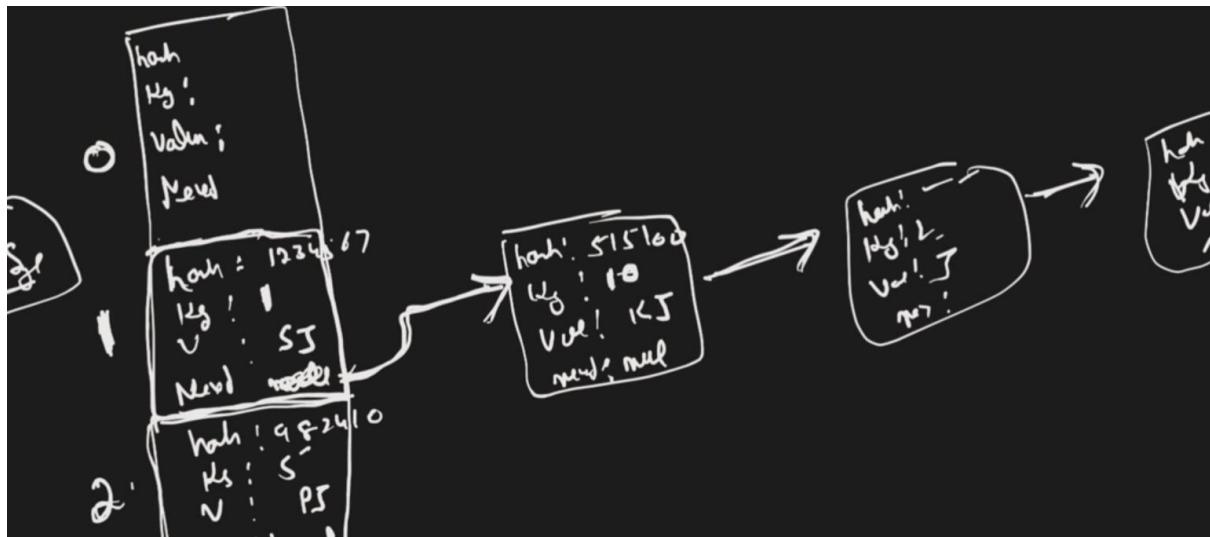
Best Case Or Average Case

Insertion,deletion,searh:O(n)

Worst

Initially until thereshold in Linked List ==O(n)

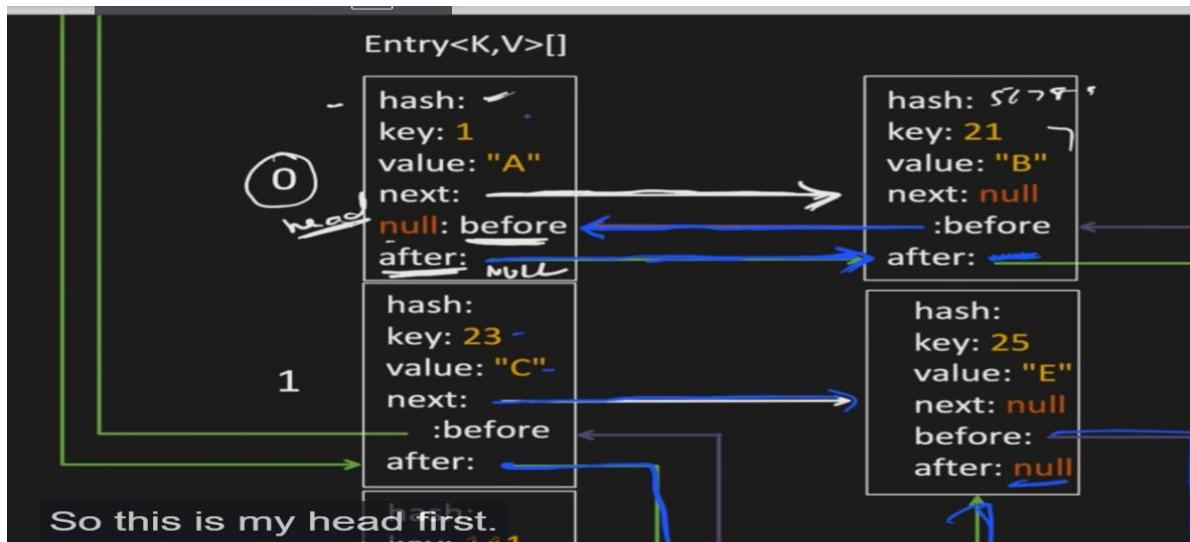
With Balanced Binary tree ==O(logn) only traverse any one side



LINKEDHASHMAP(Not Thread Safe)

Similar to hash map in terms of Architecture but

- 1) Stores insertion Order (or) Store by low Freq order to high freq order
- 2) Internally Architecture is same as Hashmap but 2 more fields will be added in addition to existing hashCode.. those are after, before to store insertion order (or) Access logic.
- 3) Uses Doubly LINKED LIST;



```
Run | Debug
public static void main(String[] args) {
    Map<Integer, String> LHP = new LinkedHashMap<>(initialCapacity:0, loadFactor:0, accessOrder:true); //access Order Freq used ele at end;
    Map<Integer, String> LHP1 = new LinkedHashMap<>(); // Normal LinkedHashMap which follows insertion order by DOUBLY LINKED LIST
    LHP.put(key:1, value:"srihas");
    LHP.putIfAbsent(key:2, value:"Srihas12");
    System.out.println(LHP);
    System.out.println(LHP.get(key:1));
    System.out.println(LHP);
}
```

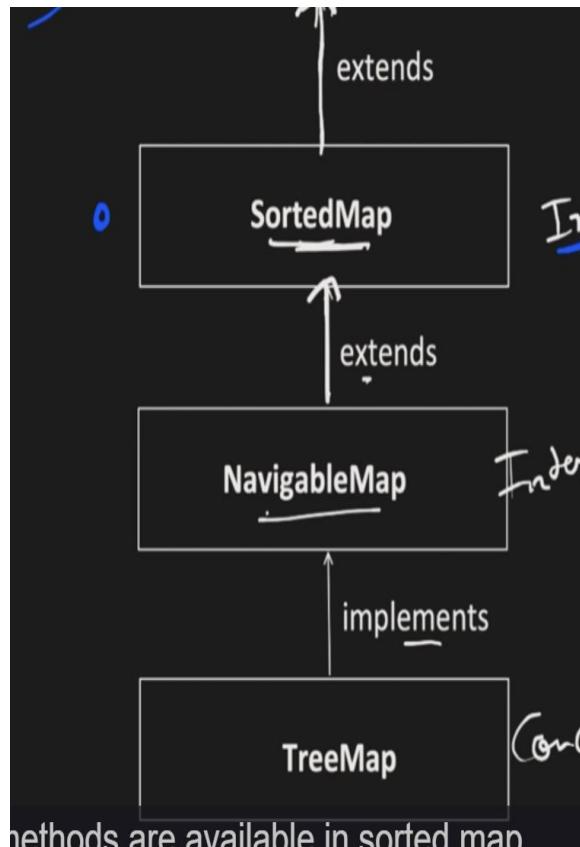
SUBCLASS MEANS EXTENDS.

TreeMap

Similar to Map but store in ascending or descending based on comparator it will store the Key.

Stores in Form of Balanced Binary Tree

Have Useful Methods



methods are available in sorted map

```
public class TreeMap1 {
    Run|Debug
    public static void main(String[] args) {
        NavigableMap<Integer, String> hp = new TreeMap<>();
        hp.put(key:1, value:"srihas");
        hp.put(key:2, value:"srihas");
        hp.put(key:0, value:"srihas");
        System.out.println(hp);
        hp.ceilingEntry(key:2);
        System.out.println(hp.descendingMap());
        NavigableSet<Integer> ns = hp.descendingKeyset();
        for (Integer integer : ns) {
            System.out.println(integer);
        }
    }
}
```

Time Complexity : O(logN)

SUBCLASS MEANS EXTENDS.

SET O(1)

SIMilar To HashMap

Store like **Entry<val,dummy obj >**

LINKEDHASHSET O(1)

Similar to LINKED HASHMAP store like DOUBLY LINKED LIST

But Store only insertion order not access order.

TREESET(=O(logn)

Similar to TreeSet

SUBCLASS MEANS EXTENDS.

MultiThreading

Process and Thread

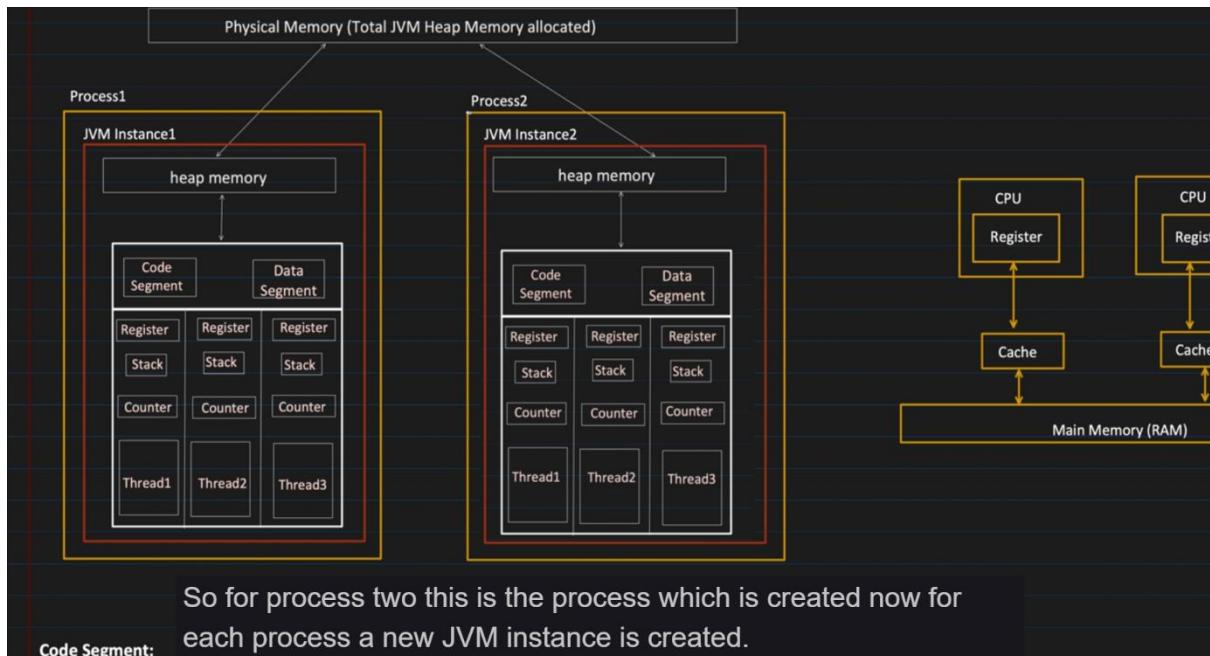
Process will have multiple Threads

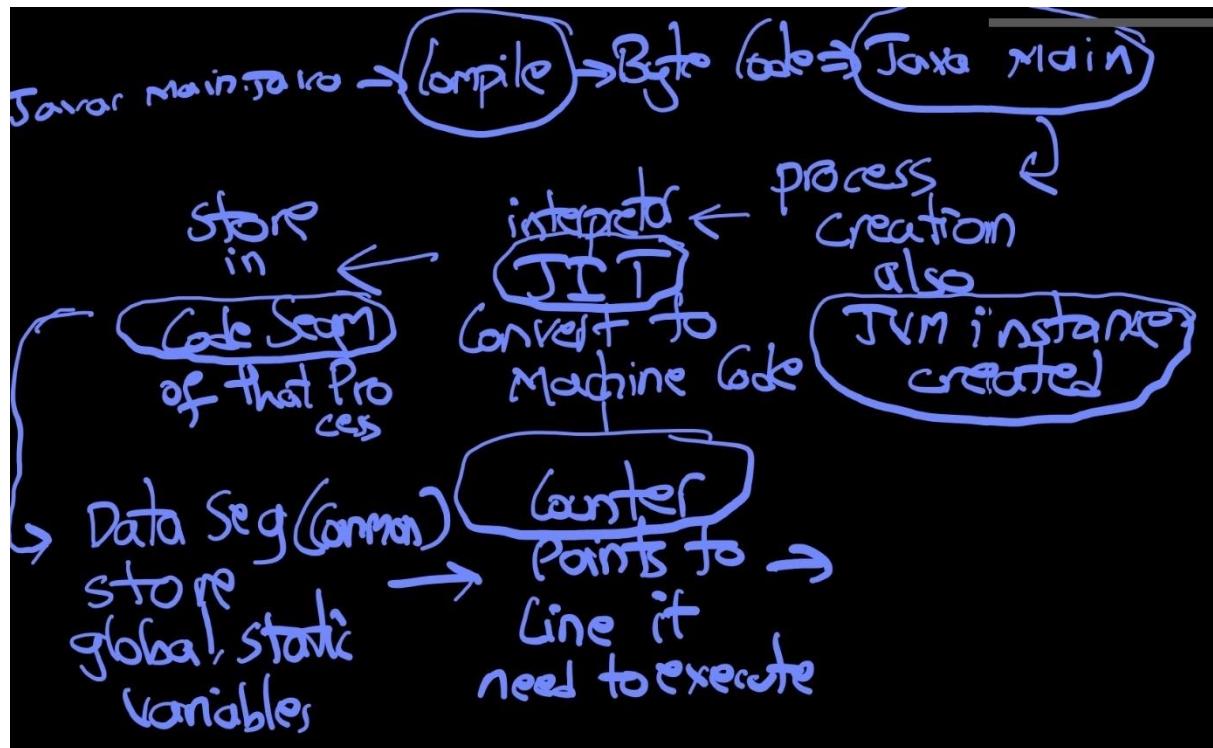
Process will be created once we run java class

Javac srihas.java → Compiled → Byte Code → java srihas → Process will be created with its own heap memory.

Initially in main it will create main thread and accordingly it will create multiple threads

Thread is a light weight process or specific set of machine understandable code. Executed by CPU





With in the specified jvm space only the **memory will be allocated** during creation of process itself we can specify size it should consume.

Compilation (`javac Test.java`) : generates bytecode that can be executed by JVM.

Execution (`java Test`) : at this point, JVM starts the new Process, here `Test` is the class which has "`public static void main(String args[])`" method.

How much memory does process gets?
While creating the process "Java MainClassName" command, a new JVM instance will get created and we can tell how much heap memory need to be allocated.

`java -Xms256m -Xmx2g MainClassName`

`-Xms<size>:`
This will set the initial heap size, above, i allocated 256MB

`-Xmx<size>:`
Max heap size, process can get, above, i allocated 2GB, if tries to allocate more memory, "OutOfMemoryError" will occur

Code Segment:

- Contains the compiled **BYTECODE** (*i.e machine code*) of the Java Program.
- Its read only.
- All threads within the same process, share the same code segment.

✓ Data Segment:

- Contains the **GLOBAL** and **STATIC** variables.
- All threads within the same process, share the same data segment.
- Threads can read and modify the same data.
- Synchronization is required between multiple threads.

Heap :

- Objects created at runtime using "new" keyword are allocated in the heap.
- Heap is shared among all the threads of the same process. (but NOT WITHIN PROCESS)
(let say in Process1, X8000 heap memory pointing to some location in physical memory, same X8000 heap memory point to different location for Process2)
- Threads can read and modify the heap data.
- Synchronization is required between multiple threads.

Stack:

- Each thread has its own **STACK**.
- It manages, method calls, local variables.

Register:

- When JIT (Just-in time) compiler converts the Bytecode into native machine code, it uses register to optimize the generated machine code.
- Also helps in context switching.
- Each thread has its own Register.

Counter:

- Also known as Program Counter, it points to the instruction which is getting executed.
- Increments its counter after successfully execution of the instruction.

All these are managed by JVM.

So this register is used right whenever the JIT uses it like

In CPU the working will be like Thread with specified register will go to CPU then OS will allocate thread to CPU and if any thread time limit reached it will use context switching and store until that part in register and when it again allocated from that point it will be executed. If threads are less and CPU is more then it is multithreading or it is just using context-switching but looks like multithreading.

Multitasking → creating new processes without common memory.

Multithreading → within Process have common memory and independent memory

SUBCLASS MEANS EXTENDS.

★

Why Stop, Resume, Suspended method is **deprecated?**

✓ **STOP** : Terminates the thread abruptly, No lock release, No resource clean up happens.

✓ **SUSPEND**: Put the Thread on hold (suspend) for temporarily. No lock is release too.

RESUME: Used to Resume the execution of Suspended thread.
Both this operation could lead to issues like deadlock.

lets see an example of it

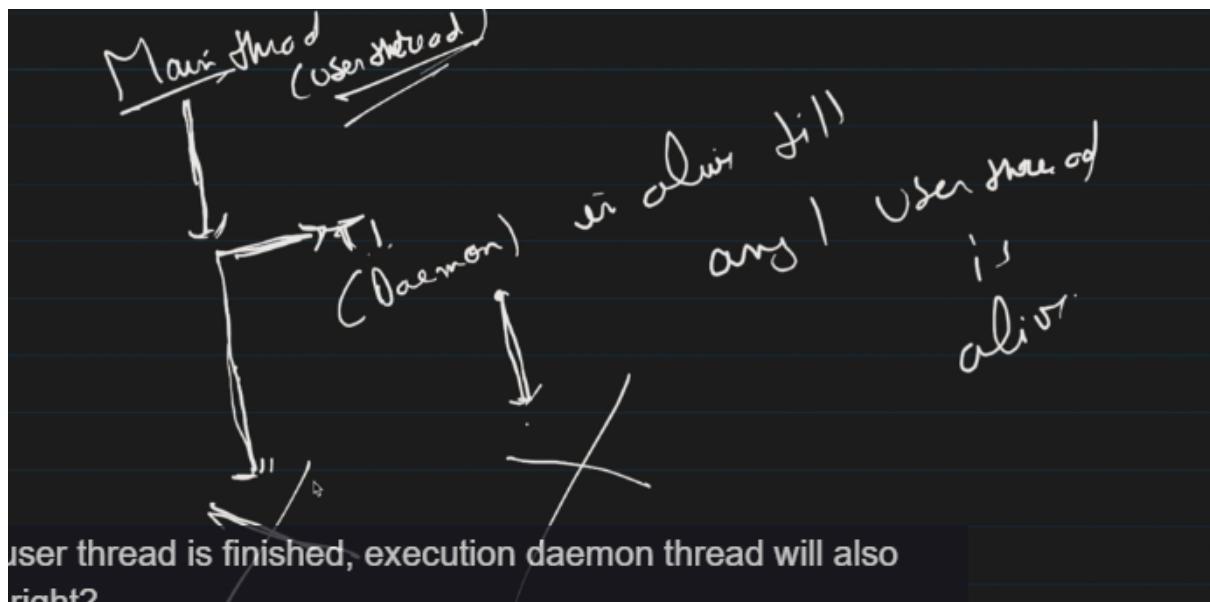
Stop and Suspend won't release locks unlike wait, so **deadlock** situation will occur so these methods are **deprecated**

Resume is used to continue the thread which is suspended hence suspend itself is deprecated resume also is.

Thread Priority

Never Guarantee that this Thread should execute 1st if it may be work or not so in production we won't use Thread priority.

Deamon Thread



Normal User Thread



Once all user Thraed are stopped Deamon Thread will be stopped unlike User Thread it wont be running even if the corresponding user thread is finished.

Synchronized work with Monitor lock

```
Obj1 {  
    Critical section  
}
```

So if both threads have Same obj then Synchronized will work because It will keep Monitor lock on obj.

But What if different Obj Trying to access same critical Section then Monitor lock and Synchronized wont work because there are multiple diff obj trying to access

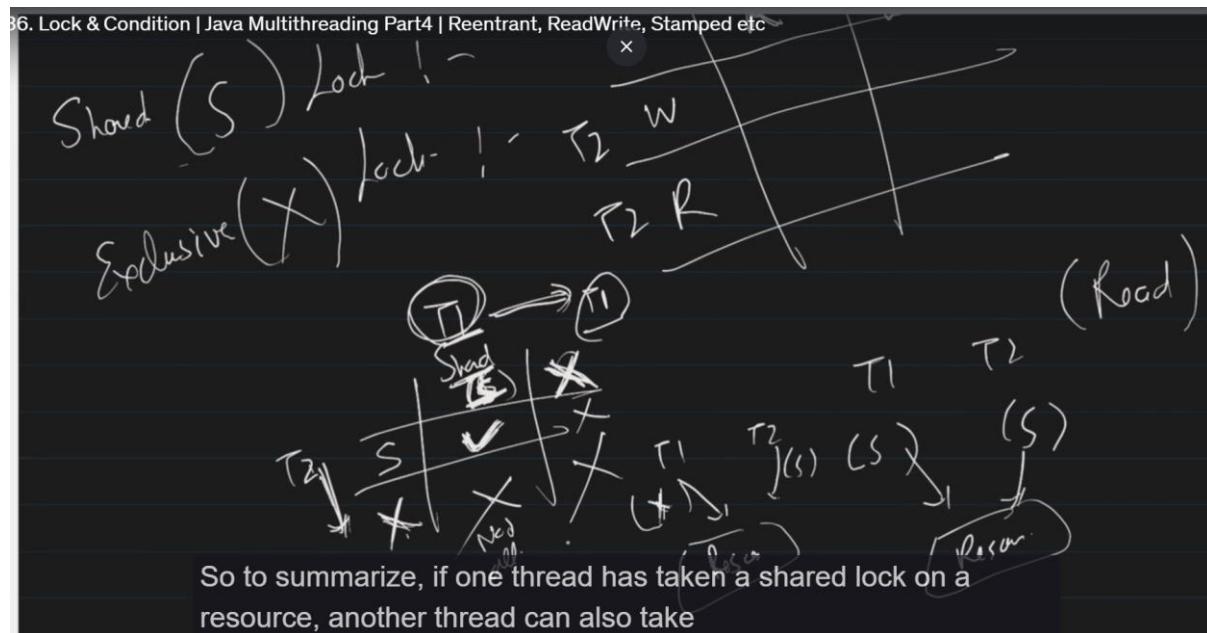
So in these Situation Custom Lock's Comes in

Types of Custom Locks

1)ReentrantLock

Wont have Synchronized but have lock and pass that lock to that function and it will work.

2)ReadWrite Lock



SharedLock → read **Exclusive Lock** → read,write

3)STAMPED LOCK

Support read,lock and also support **optimistic lock**

Optimistic Lock → Don't have lock unlock but have stamp value during locking if that stamp value is changed by other function then it wont update.

4)Semaphore Lock

Similar but can provide how many threads can access critical section

Semaphore sm =new Semaphore(2);

Atomicity

By CAS(Compare and Swap) operation

Similar to Optimistic lock where compare the version and if the version is same then only need to update values . Works on the same principle.

Don't use lock, efficient but used only in COUNTER++ like get value, compare, update

Int counter=1; Atomic, Thread safe

Class function { void counter(){count++} Not threadsafe multiple thread can come and update value so chance of data loss

Can solve by using Synchronized

Also by Atomicity

Flow

Check for value in memory by **volatile** (Means update value directly to memory not catch for data safety), if value in memory and expected is same update value in memory ,else fail and again initialize memry .

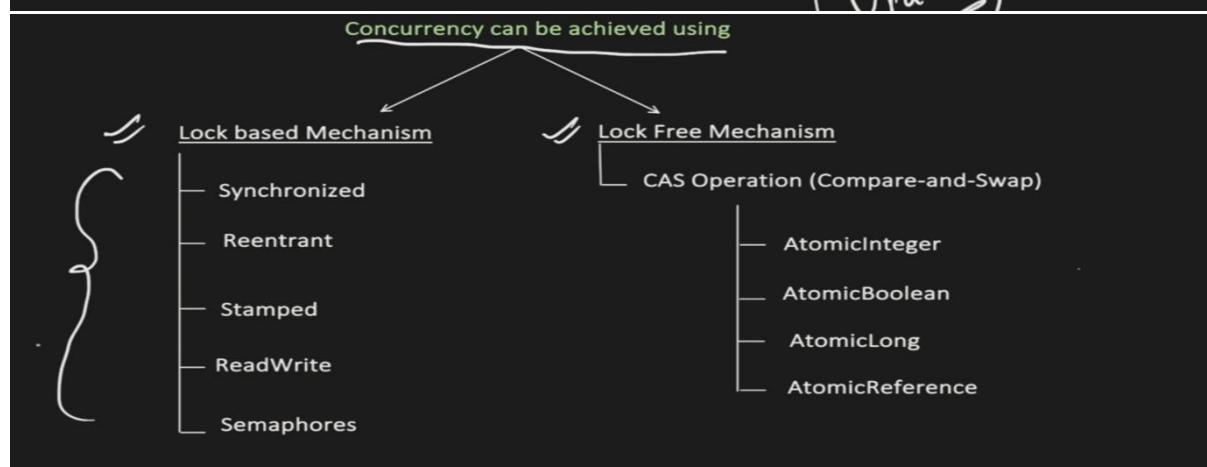
2 usages
AtomicInteger counter = new AtomicInteger(initialValue: 0);

2 usages
public void increment() {
 counter.incrementAndGet();
}

1 usage
public int get() {
 return counter.get();
}

The hand-drawn diagram illustrates a lock-free mechanism. It features a central rounded rectangle labeled "Read" at the top and "Write" at the bottom. Above it, a curved arrow labeled "Atomic" points to the "Read" section. Another curved arrow labeled "Use CAS" points to the "Write" section. A third curved arrow labeled "Wait" points from the "Write" section back to the "Read" section.

You can use atomic okay so here when we did counter plus plus right here.



It involves 3 main parameters:

- ✓ Memory location: location where variable is stored.
- ✓ Expected Value: value which should be present at the memory.
 - ABA problem is solved using version or timestamp.
- ✓ New Value: value to be written to memory, if the current value matches the expected value.

CAS (Memory, Expected Value, New Value)

X 10 9 Load / Read

ThreadPool

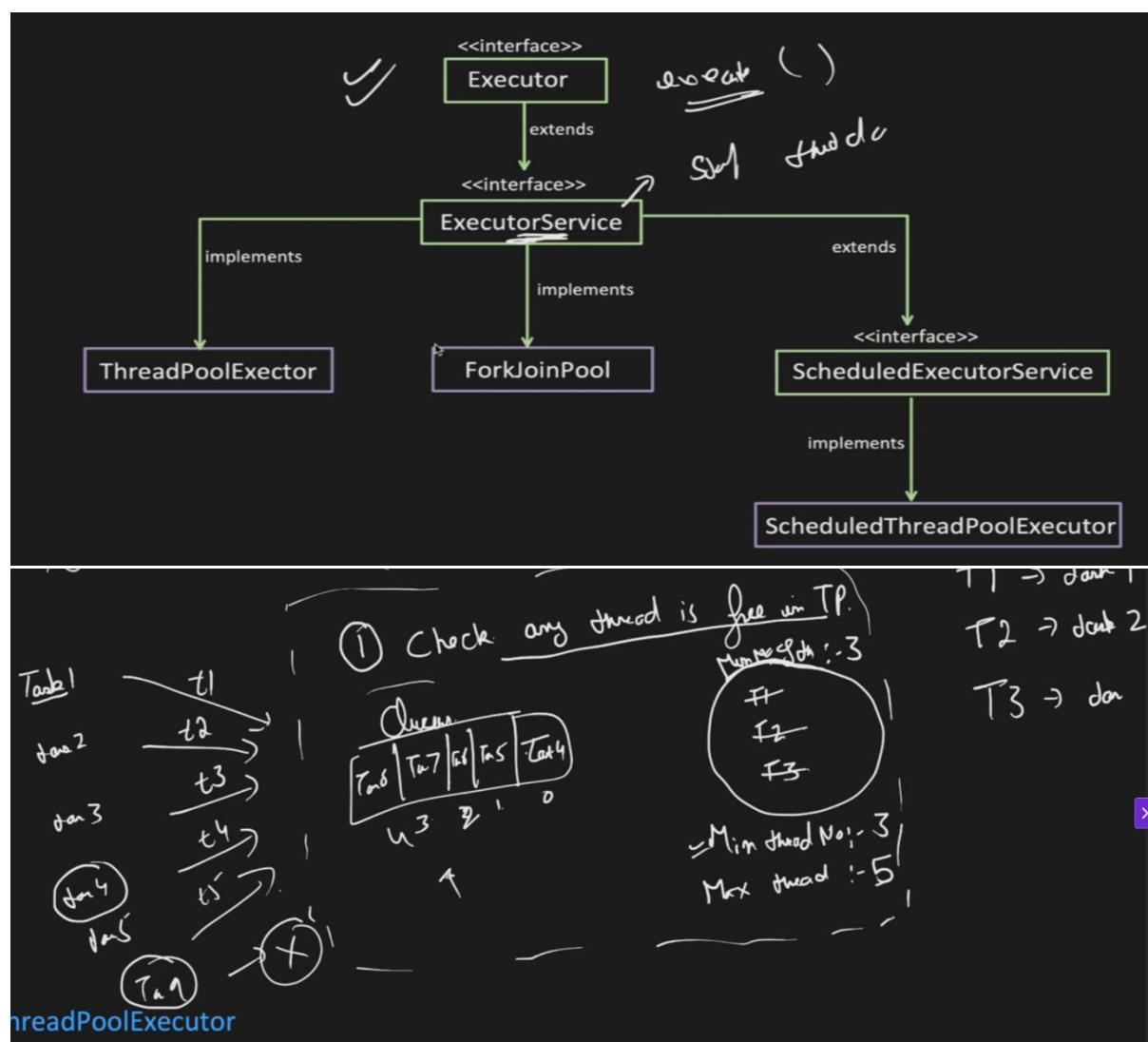
Multiple threads present inside this

Advan: 1) No need to create Thread

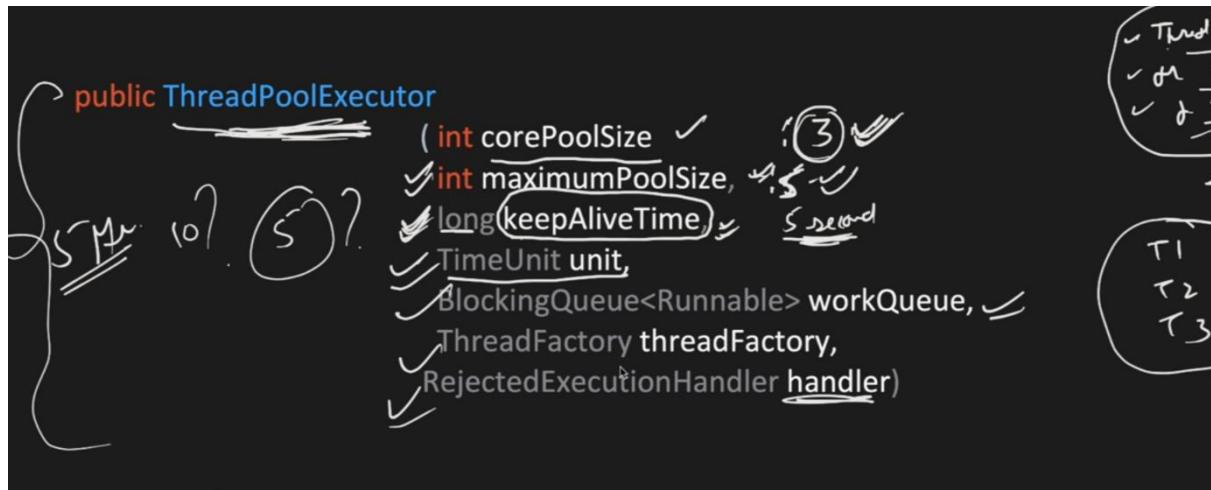
2) No Need to manage thread life cycle like create, wait, etc

3) Not create multiple threads for every task it will reuse the thread so less context switching and cpu wont be idle for longer.

How to use?



SUBCLASS MEANS EXTENDS.



CorePoolSize: min thread,

allowCoreThreadTimeout: obj where it will check for **keepAliveTime** and if it is idle more than that time it will terminate the thread. and time that we mention is defined in **units** whether it is sec, hr, min

BlockingQueue: Can be fixed Blocking queue means array, and unblocking queue like linked list, recommended to use array one.

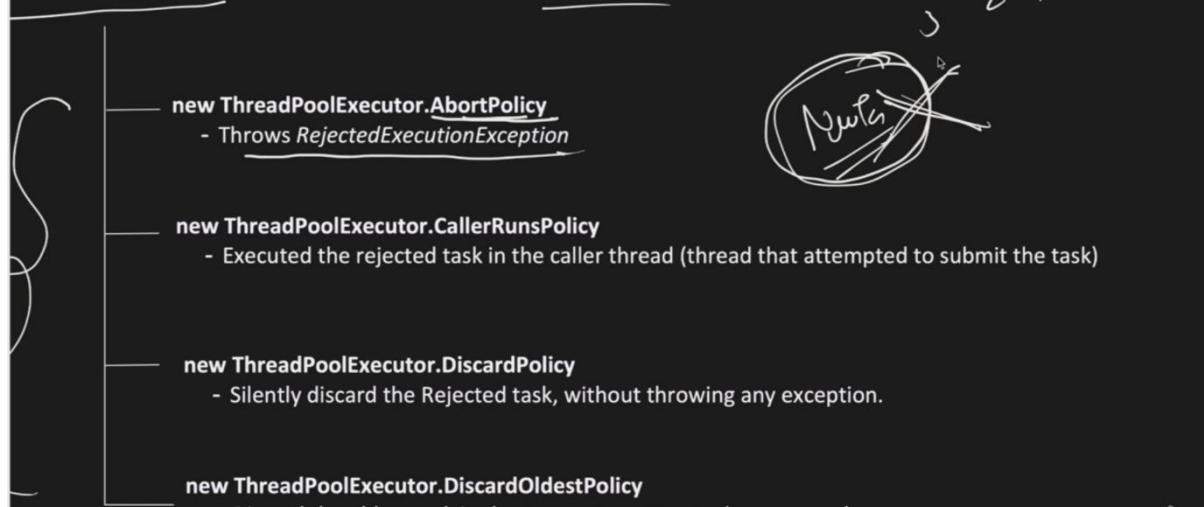
ThreadFactory: customise the thread whether name, priority and define it as demon thread etc.

RejectedExecutionHandler: To handle the rejected task if queue is full and threads are occupied they are types of it .

RejectedExecutionHandler:

handler for tasks that can not be accepted by thread pool.
Generally logging logic can be put here. For debugging purpose.

Y Y J J
z 1 0

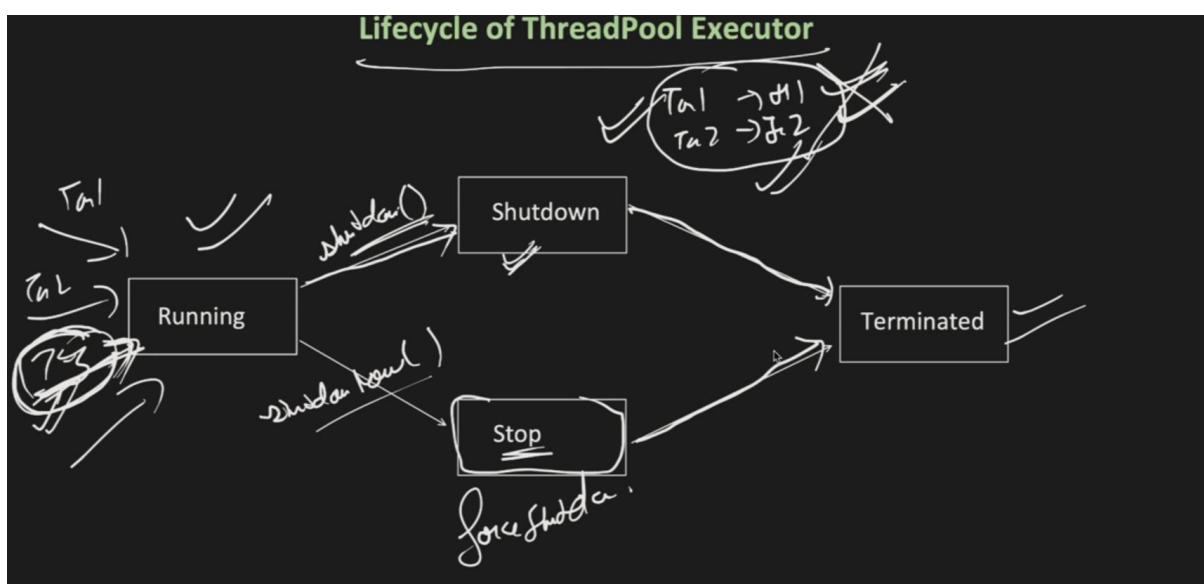


Abort policy: just throw exception

CallerRunThread: send the task to the caller like main thread to execute

DiscardPolicy: silently reject it without exception.

DiscardOldestPolicy: Remove older task in queue and replace with recent rejected task.



Running means accepts tasks, shutdown means wont accept new tasks and finish existing ones, stop even running threads.

SUBCLASS MEANS EXTENDS.

Interview question:

Why you have taken corePoolSize as 2, why not 10 or 15 or another number, what's the logic?

My Answer:

Generally, the ThreadPool min and max size are depend on various factors like:

- CPU Cores
- JVM Memory
- Task Nature (CPU Intensive or I/O Intensive)
- Concurrency Requirement (Want high or medium or low concurrency)
- Memory Required to process a request
- Throughput etc.

Max No of thread = No. of CUP Core * (1 + Request waiting time/processing time)

No. of CUP Core = 64

Request waiting time = 50ms

Processing time = 100ms

$$64 + (1 * 50/100) = \sim 64 \text{ approx.}$$

But this formula, do not consider Memory yet, which need to be consider...

JVM : 2 GB

(Heap space : 1GB

Code Cache space: 128MB

per Thread space: 5MB * N no. of threads (includes Thread Stack space)

JVM Overhead: 256MB

)

Need to consider all factors if more tasks then context switching will be more so check cpu

Then space utilization for each thread some number of space should be allocated if only 500 mb is there and each thread take 10mb then only 50 threads be created also look for heap space.

Future,Collable,CompletableFuture

Future → To know state of Thread like is it done, cancelled, wait for completion

Collable → Similar but have return type unlike Future

CompletableFuture → Implements Future have all methods and also used for chaining

By methods like thissync, thenAccept, thencompose,

Future:

- Interface which Represents the result of the Async task.
- Means, it allow you to check if:
 - Computation is complete
 - Get the result
 - Take care of exception if any etc.

```
ThreadPoolExecutor poolExecutor = new ThreadPoolExecutor(1, 1, 1, TimeUnit.HOURS, new ArrayBlockingQueue<>(10)
    Executors.defaultThreadFactory(), new ThreadPoolExecutor.AbortPolicy());

    Future<?> futureObj = poolExecutor.submit(() -> {
        try {
            Thread.sleep(7000);
            System.out.println("this is the task, which thread will execute");
        } catch (Exception e) {
        }
    });

    System.out.println("is Done: " + futureObj.isDone());
    try {
        futureObj.get(2, TimeUnit.SECONDS);
    } catch (TimeoutException e) {
        System.out.println("TimeoutException happened");
    }
    catch (Exception e) {
    }

    try {
        futureObj.get();
    }
```

{ T1 Thread
Can do

Callable:

Future<?>

- Callable represents the task which need to be executed just like Runnable.
- But difference is:
 - o Runnable do not have any Return type.
 - o Callable has the capability to return the value.

How to use this:

1. CompletableFuture.supplyAsync:

~~public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier)~~

~~public static<T> CompletableFuture<T> supplyAsync(Supplier<T> supplier, Executor~~

2. thenApply & thenApplyAsync:

✓ Apply a function to the result of previous Async computation.

- Return a new CompletableFuture object.

```
CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(() -> {  
    // Task which thread need to execute  
    return "Concept and ";  
, poolExecutor).thenApply((String val) -> {  
    // Functionality which can work on the result of previous async task
```

```
CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(() -> {
    //Task which thread need to execute
    return "Concept and ";
}, poolExecutor).thenApply((String val) -> {
    //functionality which can work on the result of previous async task
    return val + "Coding";
});
```

'thenApply' method:

- Its a Synchronous execution.
- Means, it uses same thread which completed the previous Async task.

'thenApplyAsync' method:

- Its a Asynchronous execution.
- Means, it uses different thread (*from 'fork-join' pool, if we do not provide the executor in the function*).
- If Multiple 'thenApplyAsync' is used, ordering can not be guaranteed, they will run concurrently.

3. thenCompose and thenComposeAsync:

do L
h
3

- Chain together dependent Aysnc operations.
- Means when next Async operation depends on the result of the previous.
We can tie them together.
- For aysnc tasks, we can bring some Ordering using this.

```
CompletableFuture<String> asyncTask1 = CompletableFuture
    .supplyAsync(() -> {
        System.out.println("Thread Name which runs 'supplyAsync': " + Thread.currentThread.getName());
        return "Concept and ";
    }, poolExecutor)
```

```
CompletableFuture<String> asyncTask1 = CompletableFuture
    .supplyAsync(() -> {
        System.out.println("Thread Name which runs 'supplyAsync': " + Thread.currentThread.getName());
        return "Concept and ";
    }, poolExecutor)
    .thenCompose((String val) -> {
        return CompletableFuture.supplyAsync(() -> {
            System.out.println("Thread Name which runs 'thenCompose': " + Thread.currentThread.getName());
            return val + "Coding";
        });
    });
}
```

4) thenAccept final stage don't have return type

5) thenCombine combines two CompletableFutures.

Types of Executors or Custom defined Thread Pools

'newFixedThreadPool' method creates a thread pool executor with a fixed no. of threads.

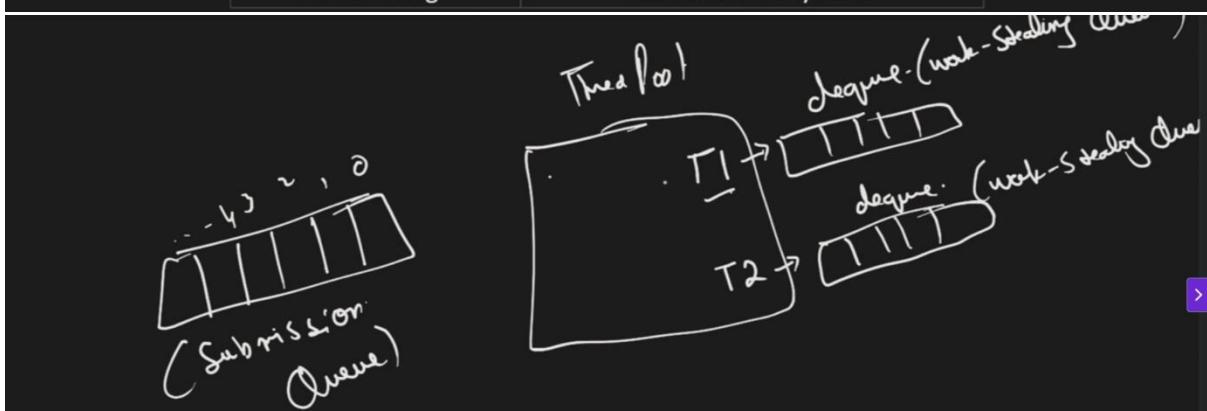
Min and Max Pool	Same
Queue Size	Unbounded Queue
Thread Alive when idle	Yes
When to use	Exact Info, how many Async task is needed
Disadvantage	Not good when workload is heavy, as it will lead to limited concurrency

```
//fixed thread pool executor
ExecutorService poolExecutor1 = Executors.newFixedThreadPool( nThreads: 5 );
poolExecutor1.submit(() -> "this is the async task");
```

<u>Min and Max Pool</u>	✓ Min : 0 Max: Integer.MAX_VALUE
<u>Queue Size</u>	<u>Blocking Queue with Size 0</u>
<u>Thread Alive when idle</u>	<u>60 SECONDS</u>
When to use	Good for handling burst of short lived tasks.
Disadvantage	Many long lived tasks and submitted rapidly, ThreadPool can create so many threads which might lead to increase memory usage.

'newSingleThreadExecutor' creates Executor with just single Worker thread.

Min and Max Pool	Min : 1 Max: 1
Queue Size	Unblocking Queue
Thread Alive when idle	Yes
When to use	When need to process tasks sequentially
Disadvantage	No Concurrency at all



Steps:

- If all threads are busy, task would be placed in "Submission Queue". (or whenever we call submit() method, tasks goes into submission queue only)
- Lets say task1 picked by ThreadA. And if 2 subtasks created using fork() method. Subtask1 will be executed by ThreadA only and Subtask2 is put into the ThreadA work-stealing queue.
- If any other thread becomes free, and there is no task in Submission queue, it can "STEAL" the task from the other thread work-stealing queue.

- Task can be split into multiple small sub-tasks. For that Task should extend:

RecursiveTask
RecursiveAction

- We can create Fork-Join Pool using "newWorkStealingPool" method in ExecutorService.

Or

By calling ForkJoinPool commonPool() method

task can be split into multiple small sub-tasks. For that

◊ RecursiveTask → return value

◊ RecursiveAction →

```
int totalSum = 0;
for (int i = start; i <= end; i++) {
    totalSum += i;
}
return totalSum;
} else {
    //split the task
    int mid = (start + end) / 2;
    ComputeSumTask leftTask = new ComputeSumTask(start, mid);
    ComputeSumTask rightTask = new ComputeSumTask(mid + 1, end);

    // Fork the subtasks for parallel execution;
    leftTask.fork();
    rightTask.fork();

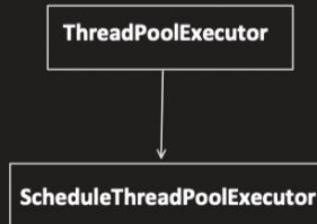
    // Combine the results of subtasks
    int leftResult = leftTask.join();
    int rightResult = rightTask.join();
```

```
public class ExecutorsUtilityExample {

    public static void main(String args[]) {
        ForkJoinPool pool = ForkJoinPool.commonPool();
        Future<Integer> futureObj = pool.submit(new ComputeSumTask(0, 100));
        try {
            System.out.println(futureObj.get());
        } catch (Exception e) {
        }
    }
}
```

ScheduledPoolExecutor

ScheduledThreadPoolExecutor : Helps to schedule the tasks



S.No.	Method Name	Description
1.	schedule(Runnable command, long delay, TimeUnit unit)	Schedules a Runnable task after specific delay. Only one time task runs.
2.	schedule(Callable<V> callable, long delay, TimeUnit unit)	Schedules a Callable task after specific delay. Only one time task runs.
3.	scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit) <pre>public static void main(String args[]) { ScheduledExecutorService scheduledExecutorServiceObj = new ScheduledThreadPoolExecutor(corePoolSize: 15); Future<?> scheduledObj = scheduledExecutorServiceObj.scheduleAtFixedRate(() -> { System.out.println("task going to start by : " + Thread.currentThread().getName()); try{ Thread.sleep(millis: 10000); }catch (Exception e){ } System.out.println("New task"); }, initialDelay: 5, period: 5, TimeUnit.SECONDS);</pre>	Schedules a Runnable task for repeated execution with fixed rate. We can use cancel method to stop this repeated task. Also lets say, if thread1 is taking too much time to complete the task and next task is ready to run, till previous task will not get
4.	scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit)	Schedules a Runnable task for repeated execution with a fixed delay (Means next task delay counter start only after previous one task completed)

Schedule:will schedule after that delay time is over

ScheduleAt FixedRate:Schedules initially after delay after that based on the period it will schedule continuously.

ScheduleAtFixedDelay: Only after completion of previous thread only delay will be done after that new schedule will happen

shutdown vs await Termination vs shutdownNow

Shutdown:

- Initiates orderly shutdown of the ExecutorService.
- After calling 'Shutdown', Executor will not accept new task submission.
- Already Submitted tasks, will continue to execute.

AwaitTermination:

- It's an Optional functionality. Return true/false.
- It is used after calling 'Shutdown' method.
- Blocks calling thread for specific timeout period, and wait for ExecutorService shutdown.
- Return true, if ExecutorService gets shutdown within specific timeout else false.

Shutdown : will not accept new tasks and finishes already running tasks.

ShutdownNow: Not finishes running task also

Await termination : check if threadpool is shutdown.

****Virtual Thread vs Normal Thread****

Normal thread or Platform Thread are the threads that we are using until now where creation on thread takes time and internally that thread is created in os and jvm wraps around it and gives access.

Disadvantages of platform or normal threads:

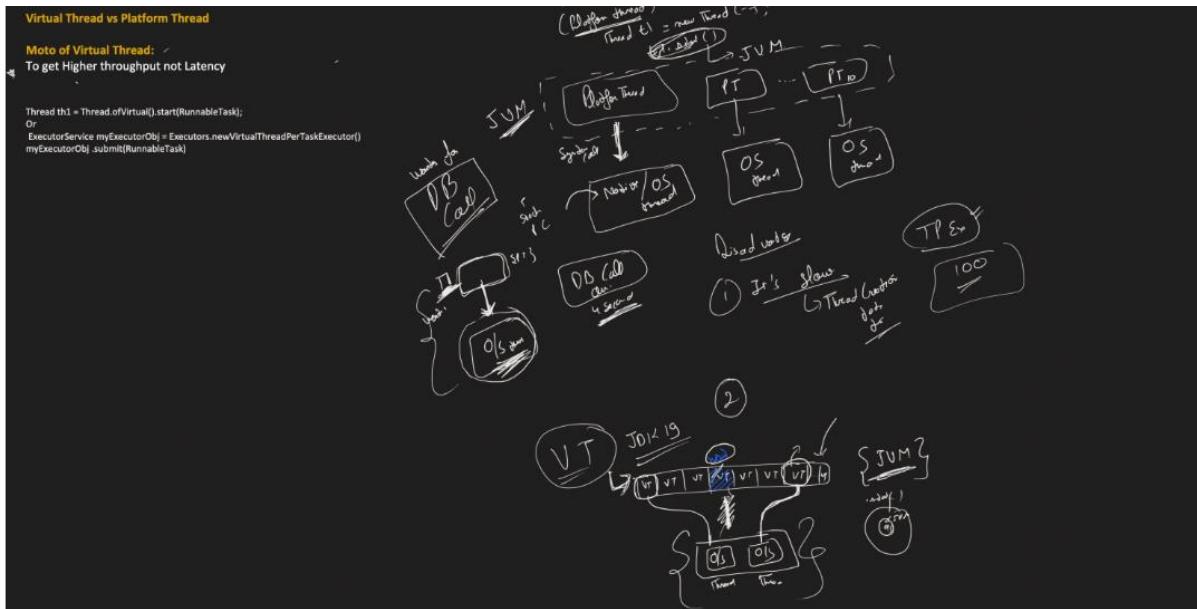
- Takes times to create Thread so we use threadPool but still inside it we need to create threads
- Also other one is once task linked to the os then if it has some db operations and goes to wait() then os will be unused .

Virtual Threads

Focus on throughput means in 1sec how many process got executed

So here we can create any number of virtual threads and those were handled by JVM

If at any point that thread goes to waiting it will unlink from os and other thread uses the OS.

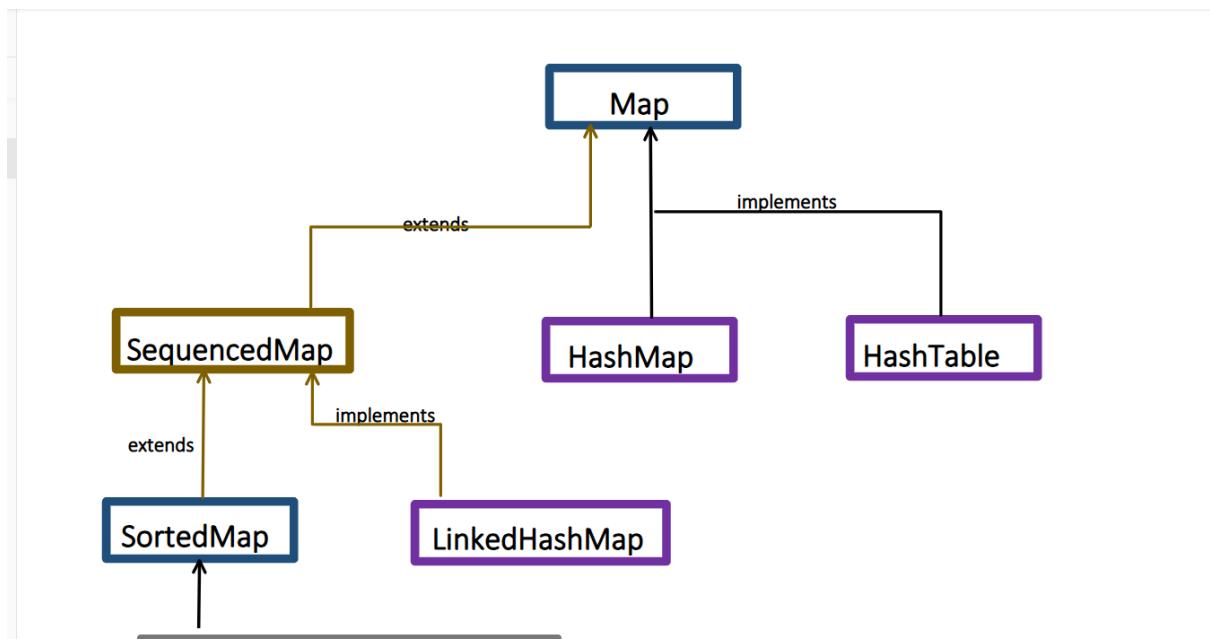
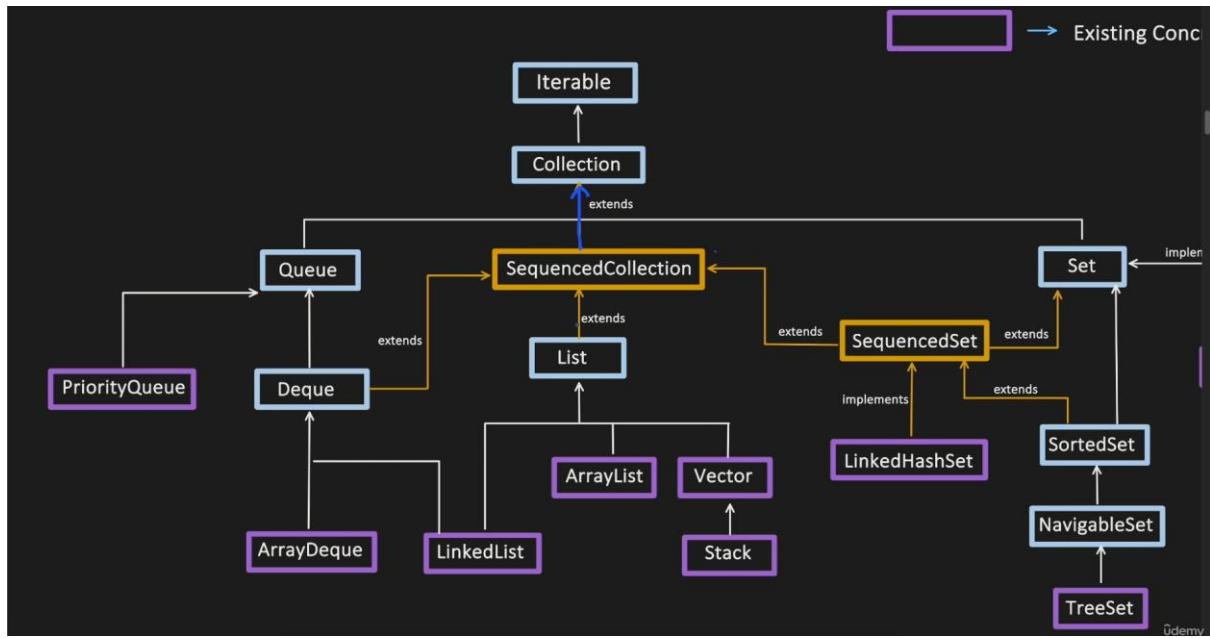


ThreadLocal

Every thread have this function Thread local we can set this and can see



SUBCLASS MEANS EXTENDS.



SUBCLASS MEANS EXTENDS.

Any Collection which follows below conditions, can be termed as **Sequenced**

- Collection should follow Predictable Iteration :

Means elements are returned in consistent and well defined order every time we iterate over the collection.

So if a collection maintains elements in:

- Insertion order, or
- Sorted order (e.g., ascending or descending)

Then we can say, it follows Predictable iteration.

- Collection provide support for Access or Manipulate the First and Last element

- Collection supports Reversible View

Like List Follows so it can be extended by sequenceCollection
but priority queue does not follow sseq so not extended
.similarly in queue can add aend so not extended.

Similalrly for Map and set also.

Why using :Using Common methods for all the sequenced collection.