Core Java

Shreyansh Jain

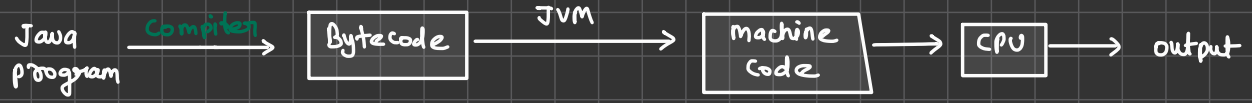
Q.1. What is java and what makes it platform independent.

→ Java is a programming language which follows OOPS concepts.
(i) Abstraction.
(ii) Inheritance.
(iii) Encapsulation.
(iv) Polymorphism.

→ Major advantage of java is its portability:- (WORA) write once run anywhere.

→ 3 main components :-

→ JVM (Java virtual Machine)
↳ abstract machine
(doesn't exist physically)



→ We get portability because of JVM. Because java code is first converted to byte code. This bytecode can be converted to machine code by JVM.

→ JVM is not platform independent. e.g. window or mac needs different JVMs.
→ Java code is platform independent.

→ JVM has JIT (Just in time) compiler

→ In Java, the just in time compiler (JIT) is the key component of Java virtual machine (JVM). Its primary function is to improve the performance of java applications by dynamically translating java bytecode into native machine code at runtime.

Here's how JIT compiler works:-

1) Interpretation:-
Initially, when a java program is executed, the bytecode generated by the java compiler is interpreted by by the JVM.
→ Interpretation involves executing each bytecode instruction one at a time.

(ii) Profiling:-
while interpreting the bytecode, the JVM collects the profiling information about the program's execution, such as frequently executed code paths, hot methods, and data types.

(iii) Optimization:-
Based on profiling information gathered, the JIT compiler identifies code segments that would benefit from optimization.
→ It applies various optimization techniques such as method inlining, loop unrolling, dead code elimination, and register allocation to improve the

performance of java application.

(iv) Compilation:-

Once JIT compiler identifies code segments that would benefit from optimization, it compiles those segments into native machine code specific to the underlying h/w architecture.

→ This native code is stored in memory and executed directly by the CPU, bypassing the interpretation overhead.

JRE (Java Runtime environment):- JVM + class libraries — default libraries.

java.math....
java.util.....

Q: Why in java single file can have only 1 public class?

→ Main method has to be inside a public class because it has to be accessed by JVM to run it as entry point.

→ Public class name should be same as file name. How will JVM know where is main method if there are more than one public classes.

Q What are variables?

variable is a container which holds a value.

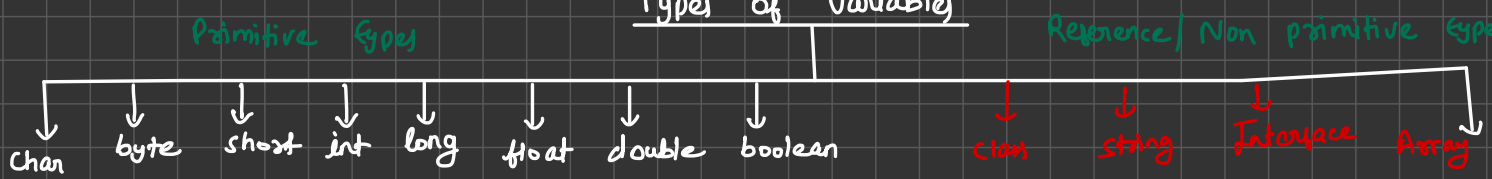
```
DataType VariableName = value;
```

→ static typed language → you have to define datatype with a variable.
→ strong typed language → for each data type we have a range b/w which we can assign values to it.

Variable naming convention:-

1). Variable name is case sensitive. e.g. int a, int A;
2). Variable name can be any legal identifier means can contain unicode letters and digits.
3). Variable name can start $, _ and letter.
4) Variable name should be all small if it contain only 1 word else camel case should be followed.
5) Variable name can't be JAVA reserved keyword like "new", "class", "while", "for" "interface", "int", "float" etc.
6) for constant, variable name should be defined in CAPITAL LETTERS

Types of variables:-



1) char :-

→ 2 bytes (16 bits)
→ character representation of ASCII values.
→ Range : 0 to 65535 i.e. '\u0000' to '\uffff'

2) byte :-

→ 1 byte (8 bits)
→ signed 2's compliment
→ Range : -128 to 127
→ default value is 0. and this works with class variable only.

3) short :-

→ 2 bytes (16 bits)
→ signed 2's compliment.
→ Range :- -32768 to 32767.
→ default value 0.

4) int

→ 4 bytes (32 bits)
→ Signed 2's compliment.
→ Range -2^{31} to $2^{31}-1$.
→ default value is 0.

5) long :-

→ 8 bytes (64 bits)
→ signed 2 compliment.
→ Range -2^{63} to $2^{63}-1$
→ default value is 0

long var = 100l;

6) float and double :-

→ 32 bit IEEE 754 value
e.g. float var = 63.20f

→ 64 bit IEEE 754 value.
e.g. double var = 63.20d;

7) boolean

→ 1 bit True/False.

→ default value = false.

→ Types of conversions :-

1) Widening / automatic conversion :-

int var = 10;
long varlog = var;

→ lower memory to higher memory automatic conversion happens.

byte (1 byte) ↘
short (2 byte) ↘ automatic conversion.
int (4 byte) ↘
long (8 byte) ↓
float
double

2) Down casting / explicit casting :- longer data type to shorter data type.

```
int var = 128;
# range of byte : -128 to 127.
byte varbyte = (byte) var;

sysо(varbyte) == -128
```

3) Promotion during expression :-

```
byte a = 127          byte sum = a+b;
byte b = 1                ↑ internal promotion to int.
```

to avoid it

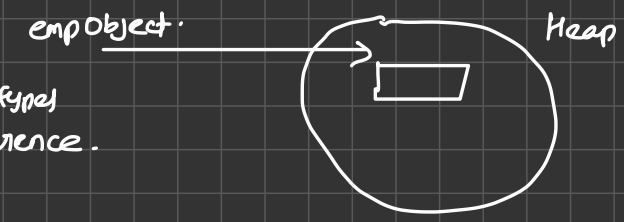
```
byte sum = (byte)(a+b);
```

↑ 4) explicit casting during expression.

Q what are reference data types ?

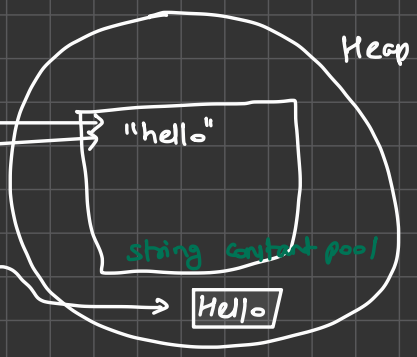
whenever you use new keyword, memory is allocated in heap.

1) Pass by value :- Java follows it.
2) Pass by reference. → Non primitive data types are passed by reference.



String :- strings are immutable.
↳ string literals.

```
String s1 = "hello";
String s2 = "hello";

String s3 = new String("Hello");
```



Interface :- public interface Person { };

Wrapper class :-

Autoboxing (change to wrapper) **Unboxing**

All primitive types have their Reference data types.

wrapper to primitive

⇒ **Constant variable :-** `static final int empId;`

Q. what is method ?

→ Method is used to perform certain tasks.
→ It's a collection of instructions that perform some specific tasks.
→ It can be used to bring the code readability and re-usability.

⇒ Access specifiers :-

Public :- can be accessed through any class in any package.

Private :- can be accessed by methods only is the same class.

Protected :- can be accessed by other classes in same package or other sub classes in different packages.

Default :- If we don't mention anything, then default access specifier is used by Java. It can be only accessed by classes in same package.

Return type :-

→ Method do not return anything use "void".
→ Use class Name or primitive data types as return type of the method

Overloaded method :-

→ More than one method with same name is created in same class.

Overridden method :-

→ sub class has the same method as parent class.

Static method :-

→ These methods are associated with the class.
→ Can be called just with class name.
→ static methods can't access non-static instance variables and methods.
→ static method can't be overridden.

when to declare method static :-
→ Methods which do not modify the state of the object can be declared
→ Utility method which do not use any instance variable and compute static. only on arguments.
e.g. factory design pattern.

⇒ Variable argument :-

→ variable number of inputs in the parameter.
→ Only one variable argument can be present in the method.
→ It should be the last argument in the list.

```
sum( int ...variable)
```

Q What are constructors and how to use it?

→ It is used to create an instance.
→ It is similar to method except :-
(i) **Name :-** constructor name is same as class name.
(ii) **Return Type :-** Constructor don't have any return type.
(iii) Constructor can't be static or final or abstract, synchronized.

(i) Default constructor:-
Java provides these constructors and assigns default values to fields.

(ii) No Argument:- className() { }

(iii) Parameterized constructor:- className(String name) { }

(iv) Private constructor:- if you want to control object creation.

→ Constructor chaining:-

(i) this (ii) super

Q. Java memory management & garbage collection.

→ 2 types of memory:-
- STACK
- HEAP

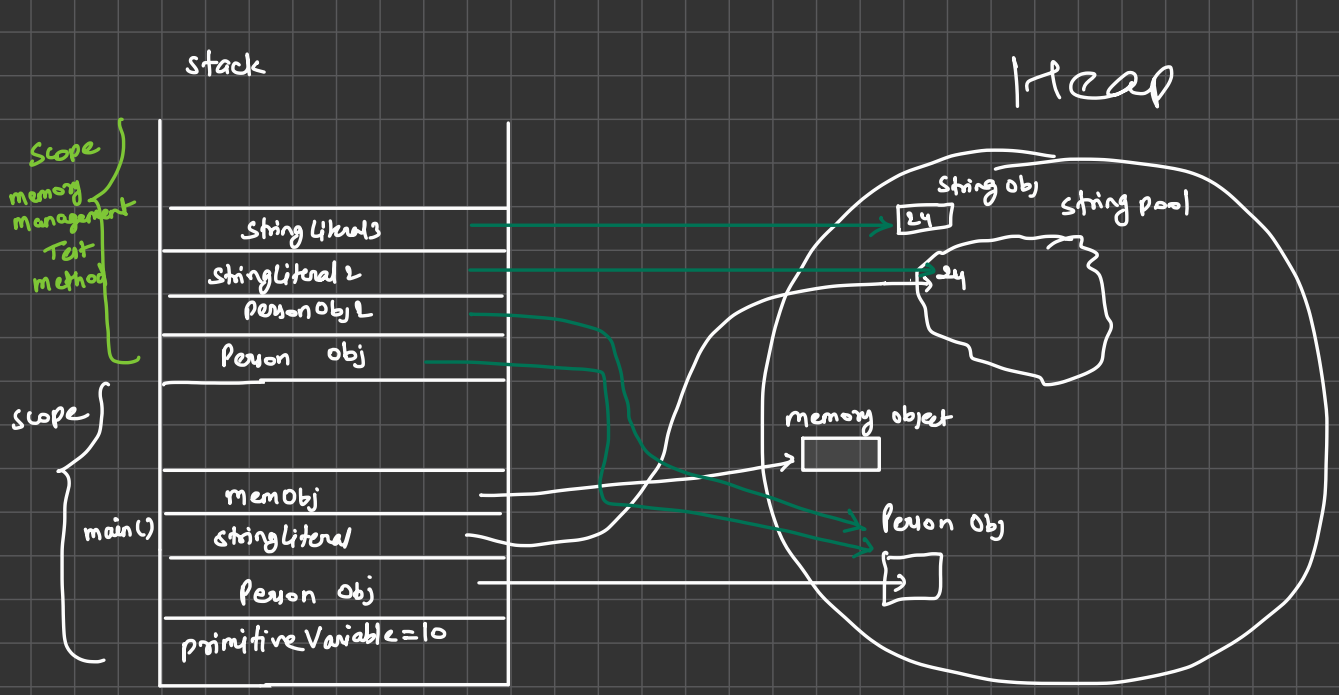
→ Both heap and stack are created by JVM and stored in RAM.

Stack memory:-

→ store temporary variables and seperate memory block for methods.
→ store primitive data types.
→ Store Reference of the heap objects.
- strong reference
- weak reference
- soft reference.

→ Each thread has its own stack memory.
→ Variables within a scope is only visible and as soon as any variable goes out of SCOPE, it get deleted. from the stack (in LIFO order).
→ when stack memory goes full, it throws "java.lang.StackOverflowError"

```
public class MemoryManagement{
    public static void main(String args[]){

        int primitiveVariable1 = 10;
        Person personObj = new Person();
        String stringLiteral = "ey";

        MemoryManagement memObj = new MemoryManagement();
        memObj.memoryManagementTest(personObj);

        private void memoryManagementTest(Person personObj){

            Person personObj2 = personObj;
            String stringLiteral2 = "ey";
            String stringLiteral3 = new String("ey");
```



→ when function ends the scope variables get deleted.

→ If nothing is referencing to memory in heap, it gets freed. by garbage collector.

→ system.gc() can scan heap for free memory. But it depends on JVM when to run it.

⇒ Different types of references:-

(i) strong reference :- Person P = new Person();
creates strong reference.

(ii) Weak reference :-

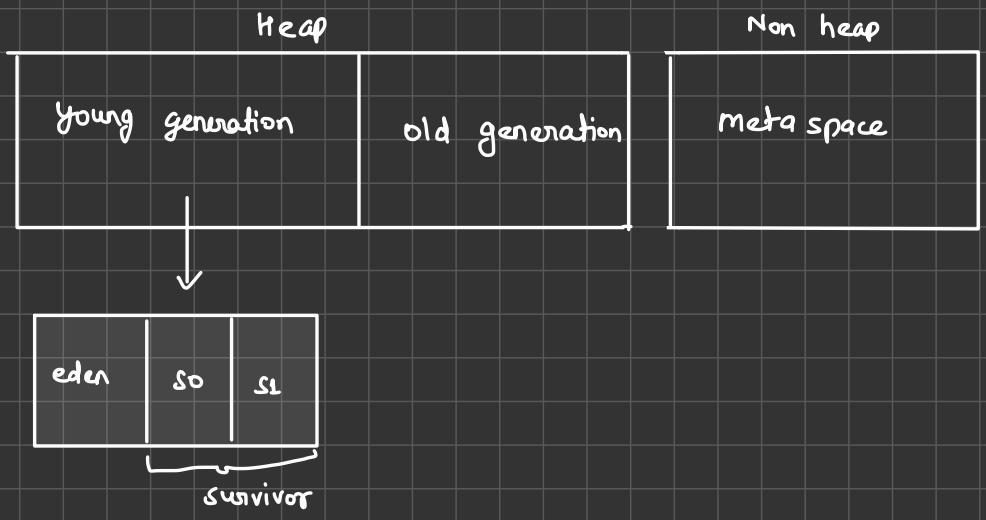
→ weak reference means whenever garbage collector runs it gets freed up.

a) Soft reference :- only free memory when it is very urgent.

⇒ HEAP memory :-

→ store objects
→ There is no order of allocating the memory.
→ Garbage collector is used to delete the unreferenced objects from the heap.
  - Mark and sweep algorithm.
  - Types of GC:
    - Single GC
    - Parallel GC
    - CMSC (concurrent Mark Sweep)
    - G1.
→ Heap memory is shared with all the threads.
→ Heap also contains the string pool.
→ When heap memory goes full, it throws "java.lang.OutOfMemoryError".

→ Heap memory is further divided into:-
- Young generation (minor gc happen here).
  - Eden
  - Survivor.
- Old generation.



→ new object goes to eden first.

① 1st iteration of GC:-
→ mark and sweep algorithm:- mark all the objects with no reference and then sweep (remove) them. from memory. from eden
And then sweep the remaining objects to s0 section. and age is applied to all the objects. And this process is called minor gc (as it happens more frequently).

② Now all new objects will again go to eden.
→ gc use s0 and s1 alternatively

→ we can set a rule that if an object reaches certain age, it is up for promotion. and that object is moved to old generation.
→ in old generation gc is called major gc because garbage collector run frequency is very less. Compared to minor gc.

→ class variables are stored in metaspace.

static variables, class metadata, constants.

⇒ Garbage collector:-

→ Mark & sweep algorithm:-
→ mark & sweep with compact memory.
↳ it puts all objects in sequential order without any space b/w them.

Q. Types of classes in Java?

→ Concrete class
→ Abstract class
→ Super class and subclass.
→ Object class.
→ Nested class
  • Inner class (Non static Nested class)
  • Annonymous inner class.
  • Member inner class.
  → local inner class
  → static Nested class (static class).

→ Generic class
→ POJO class
→ Enum class.
→ Final class
→ Singleton class
→ Immutable class
→ wrapper class.

⇒ Concrete class:-

→ These are those class that we can create an instance using NEW keyword.
→ All the methods in this class have implementation.
→ It can also be your child class from interface or extend abstract class.
→ A class access modifier can be public or package private (no explicit modifier defined).

⇒ Abstract class (0 to 100% abstraction)

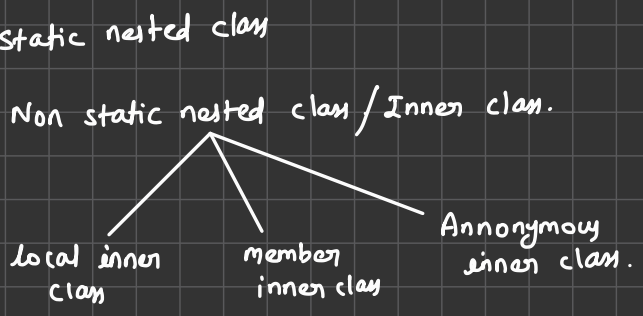
→ Show only important features to user and hide its internal implementation.
2 ways to achieve abstraction:-
- class is declared as abstracted through keyword abstract.
- It can have both abstract (method without body) and non abstract methods.
- We can't create instance of this class.
- Constructor can be created inside them. And with super keyword from child classes we can access them.

```
public abstract class Car {

    int mileage;
    Car (int mileage){
        this.
    }

    public abstract void pressBreak();
    public abstract void pressClutch();
}
```

→ Super and subclass :-

→ A class that is derived from another class is called subclass.
→ And from class through which subclass is derived its called super class.
→ Object is the topmost class in java.

→ Nested class :-



→ Class within another class.

(i) Static nested class :-

→ It do not have access to the non static instance variable and method of outer class
→ Its object can be initiated w/o initiating the object of outer class.
→ It can be private, public, protected or package-private (default, no explicit declaration).

(ii) Inner class or Non static Nested class :-

→ It has access to all the instance variable and method of outer class.
→ Its object can be initiated on after initiating the object of outer class

```
class OuterClass {
    int instanceVariable = 10;
    static int classVariable = 20;

    private static class NestedClass {
        public void print() {
            sysO( classVariable );
        }
    }
}
```

a) Member Inner class :-

→ It can be public, private, protected, default.

b) Local Inner class :-

→ These are those classes which are defined in any block like for loop, while loop block, if condition block, method etc.
→ It can't be declared as private, protected, public. Only default (not defined explicit) access modifier is used.
→ It can't be instantiated outside this block.

```
for(...){
    class Name{
    }
}
```

→ Inheritance in inner class is ponible

c) Annonymous class :-

→ An inner class w/o a name called annonymous class.

why its used :-

→ when we want to override the behavior of the method w/o even creating any subclass.

e.g.

```
public abstract class Car {
    public abstract void preBreak();
}
```

```
Car carObj = new Car() {
    @Override
    public void preBreak() {
        . . . .
    }
};
```

⇒ Generic classes :-

→ A class that can work with any data type.

→ Generic type (in above example <T>) can be any non-primitive object.

```
public class Print<T> {
    T value;
    public T getPrintValue() {
        return value;
    }
}
```

→ Inheritance with Generic class :-

1) Non Generic subclass :-

```
public class ColorPrint extends Print <String> {
}
```

2) Generic subclass :-

```
public class ColorPrint <T> extends Print<T> {
}
```

3) More than one generic type example :-

```
public class Pair <K, V> {
    . . . .
}
```

⇒ Generic method :-

→ What if we want to make method generic, not the complete class, we can write generic methods too.

→ Type parameter should be before the return type of the method declaration.

→ Type parameter scope is limited to method only.

⇒ Raw type :-

→ It's a name of the generic class or interface w/o any type argument.

→ so if we don't pass anything in <>, compiler will pass Object class here.

```
public class GenericMethod {
    public <K, V> void printValue ( Pair <K,V>
                                    pair1, Pair <K,V> pair2) {
        . . . .
    }
}
```

```
Print rawType = new Print();
```

↳ looks like `Print <Object> rawType = new Print <>();`

⇒ Bounded Generics :-

→ It can be used at generic class and method.

→ Upper bound (<T extends Number>) means T can be of type Number or its subclass only. Here super class (in this example Number) we can have interface too.

→ MultiBound :-

- <T extends superclass & interface1 & interface2>
- The first restrictive type should be concrete class.
- 2, 3, ... and so on can be interfaces.

→ Wild cards :-

- upper bounded wildcard : <? extends UpperBoundClassName> i.e. className and below.
- lower bounded wildcard : <? super LowerBoundClassName> i.e. class Name and above.
- unbounded wildcard <?> only you can read.
- we can create wildcard methods as well.

⇒ Type Erasure :-

```
public class Print <T> {
    T value;
    public void setValue(T val){
        this.value = val;
    }
}
```

Conversion →

```
public class Print{
    Object value;
    public void setValue(
                Object val){
        this.value = val;
    }
}
```

→ Generic class Bound Type Erasure :-

```
public class Print <T extends Number> {
    T value;
    public setValue(T val){
        this.value = val;
    }
}
```

```
public class Print {
    Number value;
    public void setValue(Number val)
    {
        this.value = val;
    }
}
```

⇒ POJO class :- (plain Old Java Object) :-

→ Contains variables and its getter and setter methods.
→ class should be public.
→ Public default constructor.
→ No annotations should be used. like @Table, @Entity, @Id etc.
→ It should not extend any class or implement any interface.

→ ENUM class :-

→ It has a collection of constants (variables which values can't be changed)
→ Its CONSTANTS are static and final implicitly (we don't have to write it).
→ It can't extend any class, as it internally extends java.lang.ENUM class.

→ It can implement interfaces.
→ It can have variables, constructors, and methods.
→ It can't be initiated (as its constructor will be private only, even you give default; in bytecode it make it private.
→ No other classes can extend ENUM class.
→ It can have abstract method, and all constant should implement that abstract method.

Normal enum class:-

```
public enum EnumSample{
    (0) MONDAY,
    (1) TUESDAY,
    (2) WEDNESDAY,
    (3) THURSDAY,
    (4) FRIDAY,
    (5) SATURDAY,
    (6) SUNDAY;
}
```

→ internally values of constants start from 0,1,2,.....

→ Common methods used
- values() → return array of constants.
- ordinal() → get default numbering.
- valueOf() → get value of specific constant.
- name() → get constant name

→ Enum with custom values:-

```
public enum EnumSample{

    MONDAY (101, "1st day of the week"),
    TUESDAY (102, "2nd day of the week"),
      :
      :
    SUNDAY (107, "its second weekOff");
    private int val;
    private String comment;

    EnumSample (int val, String comment){
        this.val = val;
        this.comment = comment;
    }
    public int getVal(){
        return val;
    }
    public String getComment(){
        return comment;
    }
    public static EnumSample getEnumFromValue
                                (int value){
        for (EnumSample sample: EnumSample.
                                values())
            if(sample.val == value)
                return sample;
    }
```

→ Method overriding in enums:

```
public enum EnumSample {

    MONDAY {
        @override
        public void dummyMethod
                                (){
            sysO("Monday");
        }
    },
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
      :
    SUNDAY;
    public void dummyMethod(){
        sysO("default dummy");
    }
}
```

→ abstract method can be created in enum.
→ If an abstract method is defined each constant has to override/ implement it.
→ Enum can implement an interface.

Q what is the benefit of ENUM class when we can create constant through "static" and "final" keyword?

→ enum doesn't allow any unidentified values.
→ improves readability.

→ final class can't be inherited.

Q. Singleton class in Java.

→ The class objective is to create one and only one object.

→ Different ways of creating singleton class. e.g. DBconnection.

(i) Eager Initialization :-

→ private object
→ private constructor.

```
public class DBConnection {
    private static DBConnection conObject
                = new DBConnection();
    private DBConnection(){
    }
    public static DBConnection getInstance(){
        return conObject;
    }
}
```

(ii) Lazy initialization :-

```
private static DBConnection con;
    private DBConnection(){
    }
    public static DBConnection getInstance(){
        if (conObject == null){
            conObject = new ConObject();
        }
    }
```

→ Lazy initialization has issue with multiple threads. first thread checks if condition and then second thread executes and checks if condition. so two new objects will be created.
→ solution of this is synchronized method.

(iii) Synchronized method :-

in previous code :-

→ every time this method is called thread will acquire lock on it.
→ so it is very slow so we mostly don't use it.

```
synchronized public static DBConnection
                                getInstance(){
    if (conObject == null){
        conObject = new DBConnection();
    }
    return conObject();
}
```

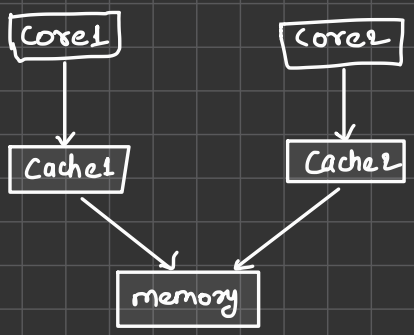
(v) Double checked locking :-

```
private static volatile DB con;
```

issue w/o volatile:-

→ In CPU there are multiple cores and each one has cache called L1 cache. And they have access to common memory.

```
public static DB getInstance(){
    if(con == null){
        synchronized (DB.class){
            if(con == null){
                con = new DB();
            }
        }
```



→ for performance core stores everything to cache and periodically sync up to the memory.

→ So if memory is not update b/w two threads check null condition then 2nd thread will find null in memory because cache didn't get updated in memory so another object will get created for DB.

(vi) Bill pugh solution:-

```
public class DB {
    private DB(){}

    private static class DBHelper{
        private static final DB OBJ = new DB();
    }

    public static DB getInstance(){
        return DBHelper.OBJ;
    }
}
```

(vii) ENUM:-

→ By default enum constructor is private.

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

Q. What are immutable class and how to make a class immutable?

→ we can't change value of an object once it is created.

→ declare class final so that it can't 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 return copy of the member variable.
e.g. String, Wrapper class etc.

Q what is interface?

Interface is something which helps 2 systems to interact with each other, w/o one system has to know the details of other.

→ Or in simple term we can say, it helps to achieve ABSTRACTION.

→ Interface declaration consists of
- modifiers
- interface keyword
- interface name.
- Comma seperated list of parent interfaces.
- Body.

→ Only public and Default modifiers are allowed (protected and private are not allowed).

```
Public interface Bird{
    public void fly();
}
```

Q Why we need interface?

1.) Abstraction:-

→ Using interface, we can define what class must do, but not how it will do.

2) Polymorphism:-

→ Interface can be used as data type.
→ we can't create the object of an interface, but it can hold the reference of all the classes which implements it. And at runtime it decide which method need to be invoked.

3) Multiple inheritance:-

→ In Java multiple inheritance is possible only through interface only.

→ Methods in an interface:-

- All methods are implicit public only.
  → methods can't be declared as final.

→ Fields in an interface:-

→ fields are public, static and final implicitly (CONSTANTS).
→ you can't make field private or protected.

→ Interface implementation:-

→ overriding a method can't have more restrict accen specifiers.
→ concrete class must override all the methods declared in the interface
→ Abstract classes are not forced to override all the methods.
→ A class can implement from multiple interfaces.

→ Nested interface:-

→ Nested interfaces can be declared within interfaces.
→ Nested interfaces declared within class.

Generally its used to group, logical related interfaces. And Nested interface

Q. Difference b|w abstract class and interface.

1. keyword used here is abstract	keyword used here is interface.
2. child classes need to use keyword extends	child classes need to use keyword implements
3. It can have both abstract and non abstract methods	It have only abstract methods.
4. It can extend from another class and multiple interfaces.	It can only extend from other interfaces.
5. Variables can be static, non-static, final and non-final.	Variables are by default CONSTANTS
6. Variables and methods can be private protected, public, default.	Variables and methods are by default public.
7. Multiple inheritance is not supported.	Multiple inheritance supported.
8. It can provide implementation of interface	It can't provide implementation of any other interface or abstract class.
9. It can have constructor.	It can't have constructor.
10. To declare a method abstract, we have to use abstract keyword and it can be protected, public default.	No need for any keyword to make method abstract. And be default its public.

Q. Java 8/9 interface methods.

→ Default method
→ static method
→ functional interface and Lambda expression.

Java 9 interface features:-

→ Private method.
→ Private static method.

→ i) Default method:-

```
default int getMin(){
    return 100;
}
```

→ To add functionality in existing legacy interface we need to use default method. e.g. stream() method in collection interface.

→ If two interfaces have same default method and one class implements both the interfaces. There will be error at compile time.
  • To resolve it we will need to provide defn of that method in class.

→ child interface can make default method abstracted.
→ child interface can use default method of parent interface by using

```
interfaceName.super.defaultMethod();
```

→ Static method of interface Java 8:-

  • We can provide the implementation of the method in interface.
  • But it can't overridden by classes which implement the interface.
  • We can access it using interface name itself.
  • its by default public.

→ private static and private methods:-

→ we can provide the implementation of method but as a private access modifier in interface.
→ It brings more readability of the code. for e.g. if multiple default methods share same code, that
→ It can be defined as static and non-static.
→ from static method we can only call private static interface method
→ Private static method can be called from both static and non static method.
→ Private abstract method can't be abstract. Means we have to provide the defn.
→ It can be used inside of the particular interface only.

Q. What is functional interface ?

→ If an interface contains only one abstract method, that is known as functional interface.

→ Also known as SAM (Single Abstract Method).

→ @FunctionalInterface keyword can be used at top of the interface but its optional.

```
@FunctionalInterface.
public interface Bird {
    void confly();
}
```

Q. What is a lambda expression ?

→ Lambda expression is a way to implement the functional interface.

→ Different ways to implement functional interface.

(i) using implements

```
public class Eagle implements
                        Bird {
    @Override
    ....
}
```

(ii) using annonymous class :-

```
Bird obj = new Bird(){
    @override
    public void confly(String val){
        ---
    }
}
```

⇒ Types of functional interface :-

(i) Consumer :-

- Represent an operation, that accepts a single input parameter and return no result.

→ present in package : java.util.function;

e.g. :-

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

(ii) Supplier :-

→ Represents the supplier of the result.

→ Accepts no input parameter but produce a result.

→ present in package : java.util.function;

```
@FunctionInterface
public interface Supplier <T>{
    T get();
}
```

(iii) Function :-

→ Represent function, that accepts one argument process it and produce a result.

→ present in package : java.util.function;

```
@FunctionalInterface
public interface Function<T, R>{
        R apply(T t);
}
```

(iv) Predicate :-

→ Represent function, that accepts one argument and return the boolean.
→ present in package : `java.util.function;`

```
@FunctionalInterface
public interface Predicate<T>{
        boolean test(T t);
}
```

→ function interface can extend to interface which has default methods but no abstract methods.

Q. what was need of functional interfaces in java?
→ Functional interfaces were introduced in java to support lambda expression
→ lambda expression allows you to treat functionality as a method argument, or to create a method body directly w/o the need to explicitly define a method.
→ They provide a concise syntax for writing anonymous functions.

→ The need for functional interfaces arose because lambda expressions require a target type.
→ A target type is a type to which a lambda expression is converted.
→ In order to use lambda expressions effectively, java needed a way to specify the target type, and functional interface serve this purpose.

→ support for lambda expressions, flexible and concise code, support for stream and APIs.

Q. what is java reflection?
→ It is used to examine the classes, methods, fields, interfaces at runtime and also possible to change the behavior of the class too.
→ for e.g.:-
- what all methods present in the class.
- what all fields present in the class.
- what is the return type of the method.
- what is the modifier of the class.
- what all interfaces the class has implemented.
- change the value of the public and private fields of the class etc...

Q. How to do reflection in class?
→ To reflect the class, we first need to get an Object of class.

what is this class Class?

→ Instance of the class Class represents classes during runtime
→ JVM creates one Class object for each and every class which is loaded during runtime.
→ This class object, has meta data information about the particular class like its method, fields, constructor etc.

⇒ How to get particular class Class object?

→ There are 3 ways:- (name of the class itself is Class)

```
class Class {
}
```

(i) using forName() method:-

→ assume we have a class Bird

```
class Bird { }
```

```
Class birdClass = Class.forName("Bird");
```

(ii) using .class :-

```
Class birdClass = Bird.class;
```

(iii) using getClass() method:-

```
Class birdClass = birdObj.getClass();
```

Now How to do Reflection:-

```
public class Eagle {
    public String breed;
    public boolean canSwim;

    public void fly() {
        sysO("Hy");
    }
    public void eat() {
        sysO("eat");
    }
}
```

```
Class eagle = Eagle.class;
sysO(eagle.getName());
sysO(Modifier.toString(eagle.getModifiers));
```

O/P = package.Eagle
public

→ Reflection of methods:-

```
Method methods[] = eagle.getMethods();   // all public methods

method[0].getName(), getReturnType(), getDeclaringClass();
```

→ Invoking methods using reflection:-

```
Class eagle = Eagle.class;

Object eagleObj = eagle.newInstance();

Method flyMethod = eagle.getMethod("fly", int.class, boolean.class, String.class);

flyMethod.invoke(eagleObj, 1, true, "hello");
```

→ Reflection of fields:-

```
Field[] fields = eagle.getFields();
    field.getName();
    field.getType();

Modifier.toString(field.getModifiers());
```

→ Setting the value of public field:-

```
Field field = eagle.getDeclaredField("breed");
field.set(eagleObj, "eagleBrownBreed");

    sysO(eagleObj.breed);
```

→ setting the value of private field:-

```
Field field = eagle.getDeclaredField("canSwim");
field.setAccessible(true);
field.set(eagleObj, true);

    sysO(field.getBoolean(eagleObj));
```

⇒ Reflection of constructors:-

```
public class Eagle {
    private Eagle(){
        . . .
    }
    public void fly(){
    }
}
```

```
Constructor[] eagleConstructorList = eagle.getDeclaredConstructors();

for(Constructor eagleCont : eagleConstructorList){

    eagleCont.setAccessible(true);
    Eagle eagleObj = (Eagle)eagleConstructor.newInstance();
    eagleObj.fly();
}
```

→ Why try to avoid using reflections?

→ It breaks to oops principle of encapsulation. You declare something private you can access it through reflection.

Q. Exception Handling in Java and its types?

→ It's an event that occurs during the execution of the program.

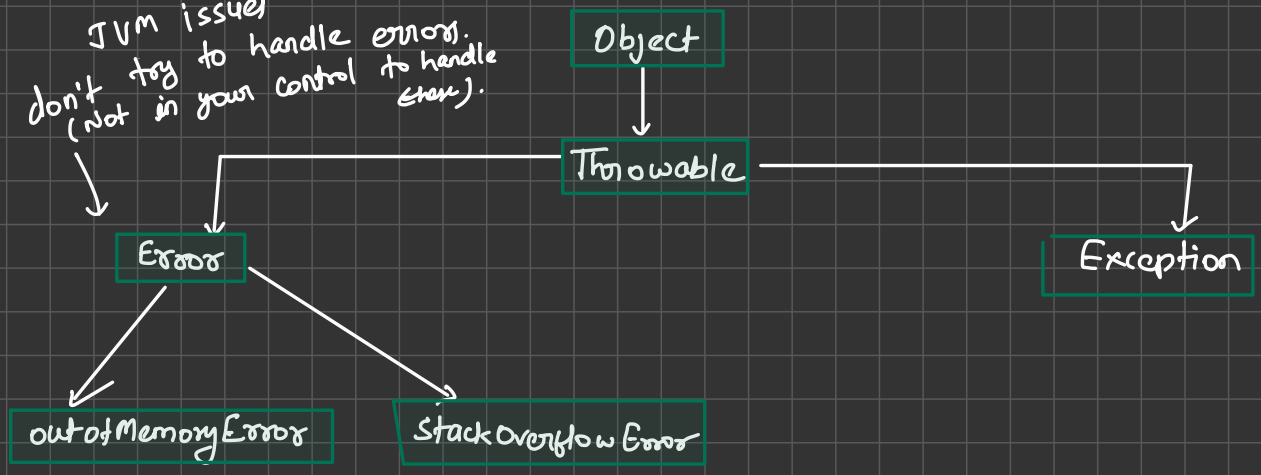
→ It will disrupt your programs normal flow.

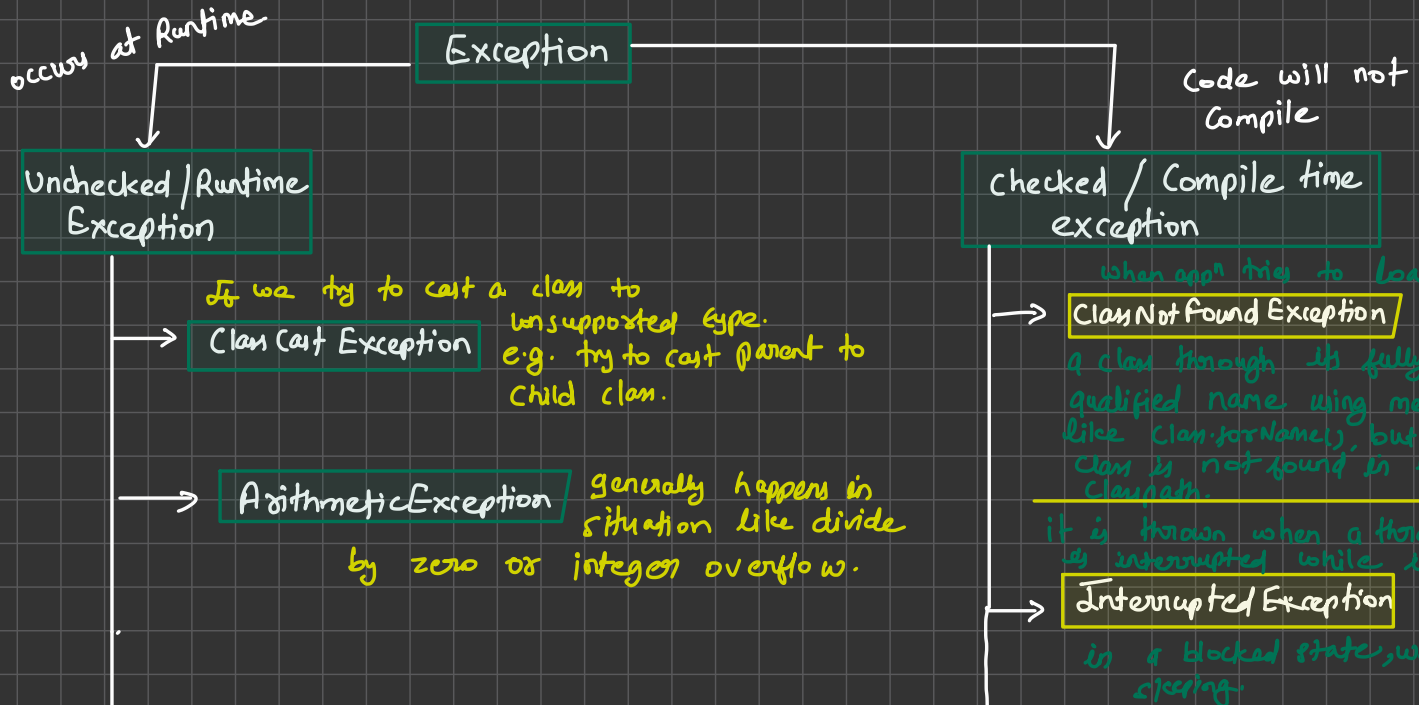
→ It creates the exception object, which contain information about the error like
- Its type of exception and message.
- Stack trace etc.

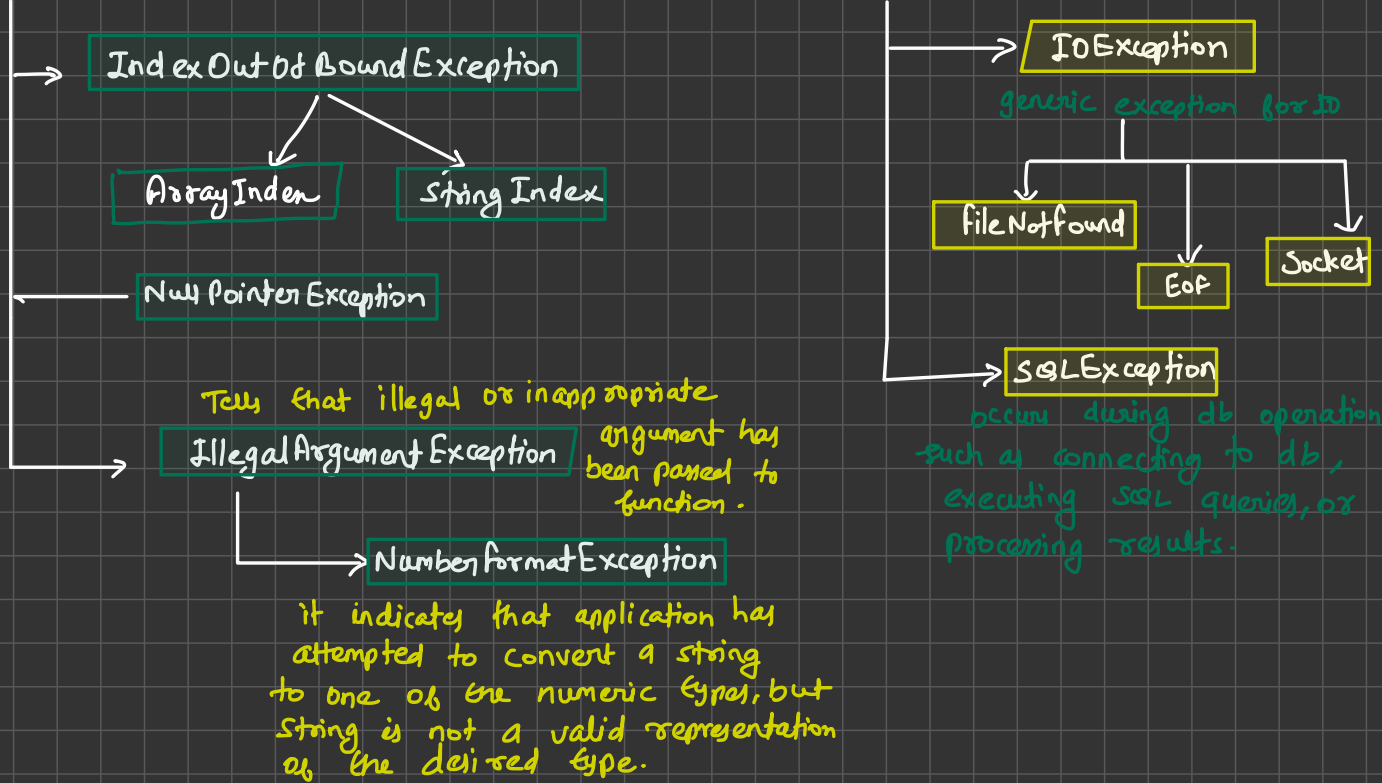
→ Runtime system use this exception object and find the class which can handle it.

→ If there is a exception Runtime system will terminate the program abruptly and print stack trace.

→ let's understand exception Hierarchy:-







→ Unchecked exception / Runtime:-

These are exceptions which occurs during runtime and compiler not forcing us to handle them.

→ Checked / Compile time exception:-

Compiler verifies them during the compile time of the code and if not handled properly, code compilation will fail.

→ Using try, catch, finally, throws:-

1) try / Catch:-

- Try block specify the code which can throw exception.
- Try block is followed either by catch block or finally block.
- Catch block is used to catch all the exceptions which can be thrown in the try block.
  → Multiple catch blocks can be used.

2) finally:-

→ Finally block can be used after try or after catch block.
→ Finally block will always get executed, either if you just return from try block or from catch block.
→ Atmost we can only add 1 finally block.
→ Mostly used for closing the objects, adding logs etc.
→ If JVM related issues like OutOfMemory, system shutdown or our process is forcefully killed. Then finally block do not get executed.

3) Throw:-

→ It is used to throw a new exception. or
- To re-throw the exception.

4) Throws :- to throw exceptions on method level.

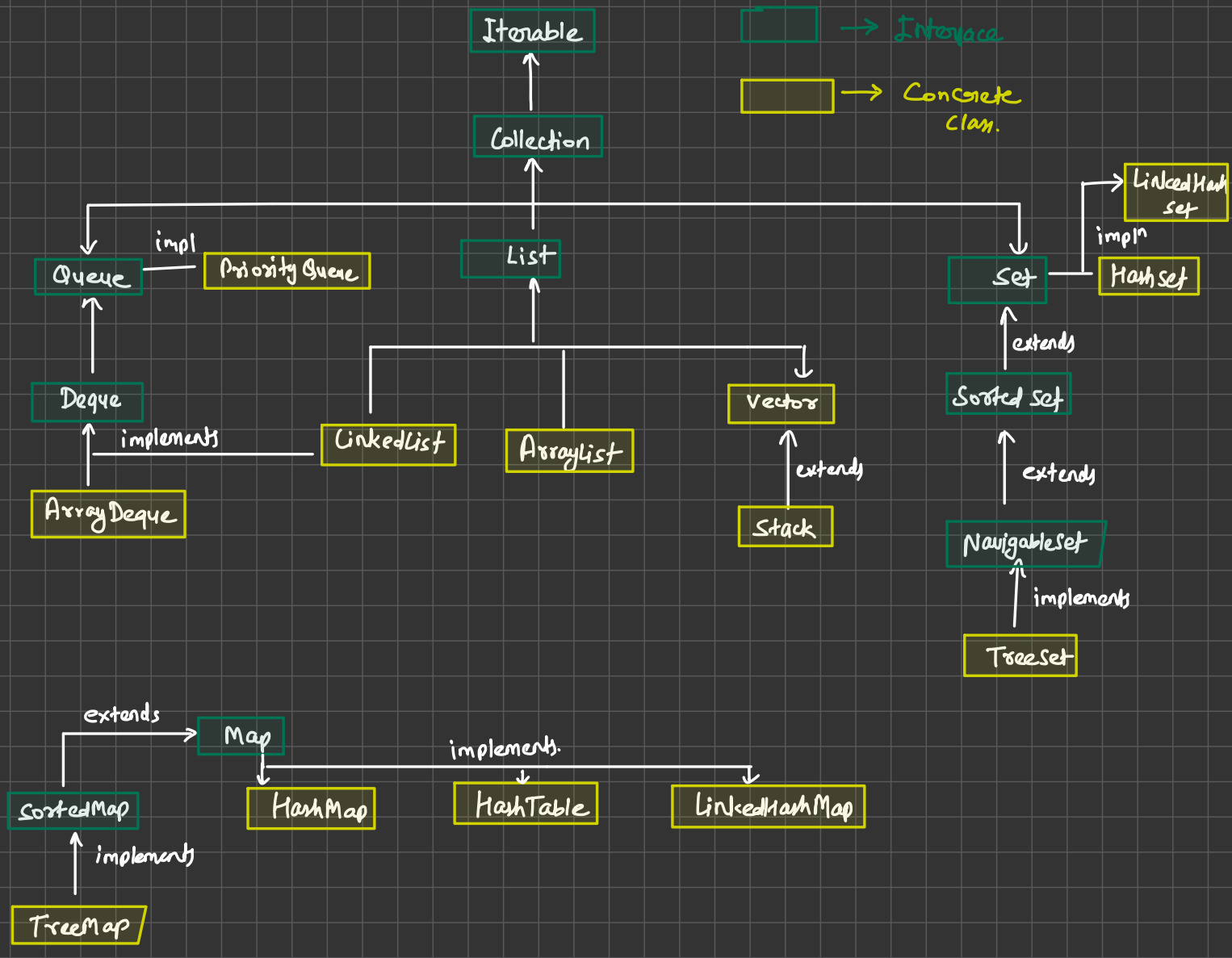
Q. what are java collections framework.

→ Added in Java 1.2.
→ Collection is nothing but a group of objects.
→ present in java.util package.
→ Framework provide us the architecture to manage these "group of objects" i.e. add, update, delete, search etc.

why we need java collections framework?

→ Prior to JCF, we have Array, Vector, and HashTable.
→ But problem with that is, there is no common interface, so its difficult to remember methods for each.

Java Collections Framework Hierarchy



(i) Iterable :- used to traverse the collection.

→ Below are the methods which are frequently used...

a) iterator() :-
it returns the Iterator Object, which provides below methods to iterate the collection.

hasNext() :- returns true, if there are more elements in collection.

next() :- Returns the next element in the iteration.

remove() :- Removes the last element returned by iterator.

b) forEach() :- Iterate collection using lambda expression. Lambda expression is called for each element of collection.

(ii) Collection :-

→ It represents the group of objects. Its an interface which provides methods to work on group of objects.

→ Common methods used by child classes :-

(i) Size :-
return the total number of elements present in the collection.

(ii) isEmpty() :-
Used to check if collection is empty or has some value. It returns true/false.

(iii) contains() :- used to search an element in the collection. returns true/false.

(iv) toArray() :- It converts collection into array.

(v) add(), remove(), addAll() :- add one collection into another.

(vi) clear(), equals(), stream() and parallelStream()

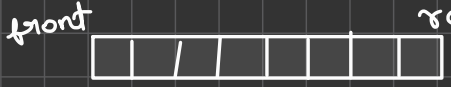
Q. Difference b/w Collection and Collections ?

→ Collection :- is part of java Collection framework. And its an interface, which expose various methods implemented by various collection classes like ArrayList, stack, LinkedList etc.

→ Collections :- is a utility class and provide static methods, which are used to operate on collections like sorting, swapping, searching, reverse, copy etc.

Q) Queue and priority Queue ?

→ Queue is an interface, child of collection interface.
→ Generally queue follows FIFO approach, but there are exceptions like PriorityQueue.
→ Supports all the methods available in collection + some other methods mentioned below.



1.	add()	- True if insertion was successful and exception if insertion fails. - Null element insertion is not allowed will throw (NPE)
2.	Offer()	- True if insertion successful, False if fails. - Null element insertion is not allowed will throw (NPE).
3.	Poll()	- Retrieves and removes the head of queue. Returns null if queue is empty.
4.	remove()	Retrieves and removes head of the queue. Returns Exception (NoSuchElementException) if queue is empty.
5.	Peek()	Retrieves the value present at the head of the queue but do not remove it. Returns null if queue is empty.
6.	element()	Retrieves the value present at the head of the queue but don't remove it. - Returns an Exception (NoSuchElementException). if queue is empty.

PriorityQueue :-

→ Its of two types min priority queue, and maxm priority queue.
→ It's based on priority Heap (Min Heap and Max Heap).
→ Elements are ordered according to either natural ordering (by default) or by comparator provided during queue construction.

Time Complexity :-

→ Add and Offer : O(logn)
→ Peak : O(1)
→ Poll and remove head element: O(logn)
→ Remove arbitary element: O(n).

Q. Comparator and Comparable ?

→ Both provide a way to sort the collection of objects.

(i) Comparator:-

```
int compare(T obj1, T obj2)
```

→ sorting algorithm uses this compare method of Comparator to compare two variables and decide whether to swap the variables or not.

→ Method returns:-

1: if v1 is bigger than v2.
0: if v1 and v2 is equals.
-1: if v1 is smaller than v2.

→ Mostly in this algorithm, if this method return 1, it swap the values.

(ii) Comparable:-

```
int compareTo(T obj2);
```

→ Sorting algorithm uses this compareTo method of Comparator to compare variables and decide whether to swap or not.

Q. what is deque ?

Interface.

Deque (Double ended queue) :- addition and deletion can be done on both sides of the queue.

→ Methods available in deque:-

Methods available in Collection, Queue interface + Methods available in deque interface.

Queue:- add(), offer(), poll(), remove(), peek(), element()

	Throw Exception	Do not throw	Throw	donot
Insert Operations	addFirst(e)	OfferFirst(e) return true/false	addLast(e)	offerLast() return true
Remove Operations	removeFirst()	pollFirst() return null if deque empty	removeLast()	pollLast() return null
Examine Operations	getFirst()	peakFirst() return null if empty.	getLast()	peakLast return null.

→ using these methods, we can even use deque to implement stack (LIFO) and Queue (FIFO) both.
→ To use it as stack push and pop methods are available.
→ push internally calls addFirst().
→ pop internally calls removeFirst().

ArrayDeque :- (concrete class).

→ implements the methods which are available in queue and deque interface.

```
ArrayDeque<Integer> dArray = new ArrayDeque();
```

→ **Time complexity:-**

Insertion:- Amortized (Most of the time or Average) complexity is O(1) except few cases like O(n): when queue size threshold reached and try to insert an element at the end or front, then its O(n) as values are copied to new queue of bigger size

Deletion:- O(1)

Search :- O(1)

space complexity:- O(n).

Collection	isThreadSafe	Maintains Insertion order	Null Elements allowed	Duplicate elements allowed
priority queue	NO	NO	NO	YES
Array Deque	NO	YES	NO	YES

PriorityBlockingQueue:- thread safe priority queue.

```
PriorityBlockingQueue<Integer> pq = new PriorityBlockingQueue();
```

ConcurrentLinkedDeque:- thread safe

```
ConcurrentLinkedDeque<Integer> ob = new ....<>();
```

Q. what is a List? How to use it.

→ List is an ordered collection of an object. In which duplicate values can be stored.

→ In list user can decide where to insert or access using index (starting from 0).

→ ListIterator<E> listIterator() :- this exposes some extra methods than Iterator.

e.g. hasPrevious(), previous(), nextIndex(), previousIndex(), set(E e), add(E e).

ArrayList:- Concrete class. implementation of List.

LinkedList:- implements deque and list.

Vector:- Concrete class for List.

→ Exactly same as arraylist; elements can be accessed via index
→ It's thread safe
→ Puts lock when operation is performed on vector.
→ Less efficient than ArrayList as for each operation it do lock/unlock internally.

Stack:- Child of vector. so methods are synchronized.

→ It can be implemented using deque, but deque methods are not thread safe.

Q What is map and how does it work internally?

Why Map is not child of Collection?
→ Methods in Collection deal with single value, while in map methods should deal with key-value pairs.

⇒ **Map properties:-**

→ It's interface and its implementations are:
- HashMap: do not maintain the order.
- HashTable: Synchronized version of HashMap.
- TreeMap: sorts the data internally.

- Object that maps key to values.
- Can't contain duplicate key.

⇒ Methods available in Map interface:-
e.g. size(), isEmpty(), containsKey(), containsValue(), get(Object key), put(K key, V val), remove(Object key), putAll(Map<K,V> m), clear(),

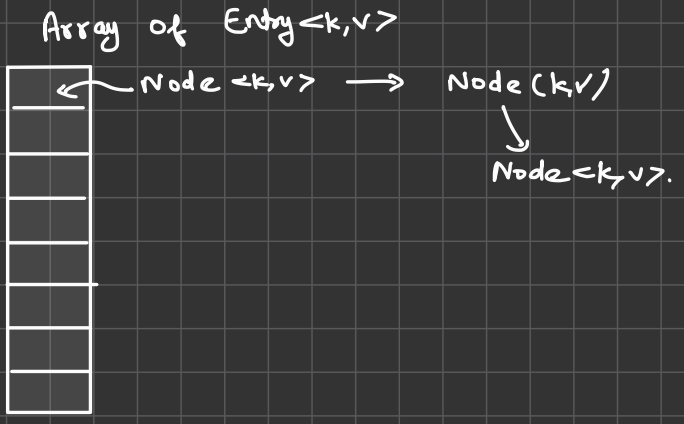
Set<K> keySet(), Collection<V> values(), Set<Map.Entry<K,V>> entrySet, putIfAbsent(K key, V value), getOrDefault(key, defaultValue), Entry sub-interface

⇒ **HashMap:-**

→ Can store null key or value (HashTable doesn't contain null, key or value).
→ It doesn't maintain insertion order.
→ It's not thread safe (instead we ConcurrentHashMap or HashTable for ThreadSafe HashMap implementation).

→ Node <k,v> implements Entry interface.

```
Node {
    final int hash;
    final K key;
      V value;
    Node<k,v> next;
}
```



→ default capacity is 16 for Hashmap

→ when put(key, value) method is called it performs different tasks.

HashMap <Integer, String>

(i) First compute Hash for key. e.g. MD5, SHA256 hashing algo examples.

```
index = hashValue % sizeOfTable
```

→ if collission happens on index
(i) check if key is same (hash is same or not).
(ii) if not it will create a new node with new key value and next will point to new Node.

→ For same element you should get the same hashCode every time we use hash function.

→ Because there is a contract b/w hashCode & equals method
① first contract:- if obj1 == obj2 then their hash should also should be same.
② second contract:- if two objects have same hash that doesn't mean they are same objects.

e.g. Hash is equal for elements on same index, but element might be somewhere else in linked list. so objects are not same for same hash value.

① load factor : default is 0.75, 16 x 0.75 = 12

→ so if hashmap gets 12 indexes full, it will do rehashing and increase double the size of array.

② Treefy Threshold:- default is 8.

→ Once the linked list has 8 elements it will convert linked list to Balanced Binary Search Tree. (Red/Black) tree.

→ Searching will be O(log n) Binary search.

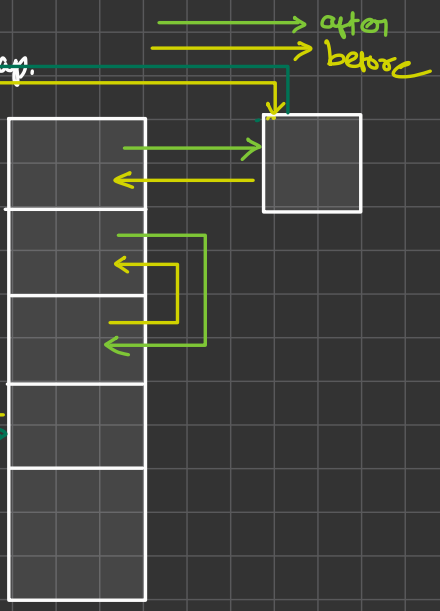
HashTable:- It is synchronized version of HashMap.

⇒ Linked HashMap:-

→ Helps in maintaining insertion order.
→ Similar to HashMap, but also uses double linked list.

Node has extra elements for Linked HashMap.

```
Node {
        hash
        key
        value
        before
        After
    } next
```



→ Stores Linked list process is same as HashMap for put and get.

Access order
→ elements which are accessed frequently are at the last and less frequently accessed elements are at start.

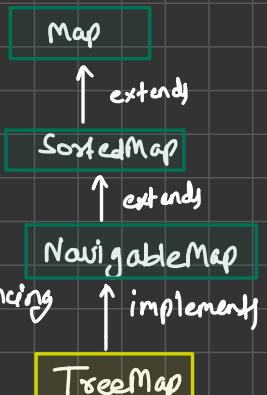
→ by default access order is false. That means you want insertion order

```
Map<Integer, String> map = new LinkedHashMap<>(initialCapacity:16,
                                loadFactor:.75F, accessOrder:true);
```

→ It's not thread safe and no thread safe version is available for this.
→ so we have to explicitly make this collection thread safe like this:

```
Collections.synchronizedMap(new LinkedHashMap());
```

⇒ Treemap:- (concrete class).



→ Map is sorted according to its natural ordering of its keys or by comparator provided during map creation
→ It is based on Red-Black tree (self balancing Binary Search Tree).
→ O(log n) time complexity of insert, remove, get operations.

Q. Explain working of sets in Java.

→ Few properties of set:-

1) Collection of objects, but it doesn't contain duplicate value (only one null value you can insert) and

2) Unlike list, set is not an ordered collection, means objects inside set doesn't follow the insertion order.

3) Unlike list, set can not be accessed via index.

⇒ Few questions:-

(i) What data structure is used in set internally (as it doesn't allow duplicate values).
(ii) As order is not guaruntee, then what if we want to sort set?

⇒ What all methods set interface contains:-

→ All methods from Collection interface, generally that only is available in set interface. No new method specific to set added.

HashSet:-

→ Datastructure used: HashMap.

```
HashMap<E, Object> map = new HashMap<>();
```

→ During add method invocation, it stored the elements in key part and in value it stores the dummy object.

```
map.put(element, new Object());
```

→ in set only one dummy object is used.

```
private static final Object PRESENT = new Object();
```

→ No guaruntee that the order will remain constant.

→ HashSet is not threadSafe. newKeySet method present in ConcurrentHashMap class is used to create thread safe set.

⇒ LinkedHashSet:-

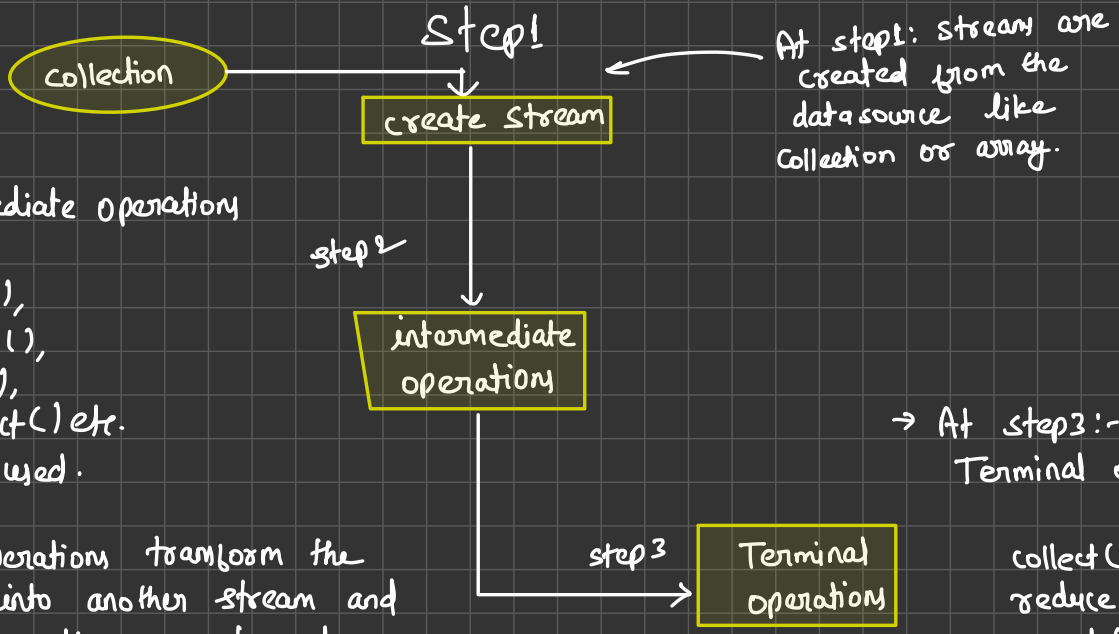
→ Internally uses: LinkedHashMap
→ Maintains the insertion order of the element.
→ It is not thread safe.

API Streams

Q what is stream and what features it provide?

→ we can consider stream as a pipeline, through which our collection element pass through.
→ while elements pass through pipelines, it performs various operations like sorting, filtering etc.

→ useful when deals with bulk processing. (can do parallel processing)



→ These operations transform the stream into another stream and more operation can be done on top of it.

→ These are lazy in nature, means these operation gets executed only when terminal operation is invocked.

→ These operation trigger the processing of stream.
→ And produce the output.
Means after terminal operation used, no more operation we can perform.

⇒ Different ways to create a stream:-

1.) From Collection:-

```
List<Integer> salaryList = . . .

Stream<Integer> stream = salaryList.stream();
```

2). From Array:-

```
Stream<Integer> stream = Arrays.stream(salaryArray);
```

3.) From static method:-

```
Stream<Integer> stream = Stream.of(10, 20, 30, 40);
```

4) From Stream Builder:-

```
Stream.Builder<Integer> streamBuilder = Stream.builder();
streamBuilder.add(1000).add(9000).add(3500);

Stream<Integer> stream = streamBuilder.build();
```

5) From stream Iterate:-

```
Stream<Integer> stream = Stream.iterate(1000, (Integer n) -> n + 500).limit(5);
```

Parallel stream:-

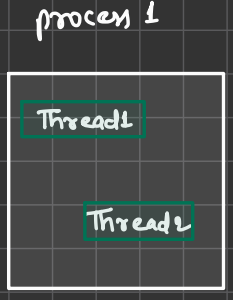
→ Helps to perform operation on stream concurrently, taking advantage of multi core CPU.
→ parallelStream() method is used instead of regular stream() method.

→ Internally it does:-
  → Task splitting:- it uses "splitIterator" function to split the data into multiple chunks.
  → Task submission and parallel processing: uses fork join pool technique.

Multithreading and Concurrency in Java

→ Before multithreading let's first understand process and thread.

Process:-



→ process is an instance of a program that is getting executed.

→ It has its own resources like memory thread etc. OS allocate these resources to process when its created.

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(...)" 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, here allocated 256MB.

-Xmx <size>

Max heap size, process can get, above, allocated 2GB here, If it tries to allocate more memory, "OutOfMemoryError" will be thrown.

⇒ Thread

→ Thread is known as lightweight process.

OR

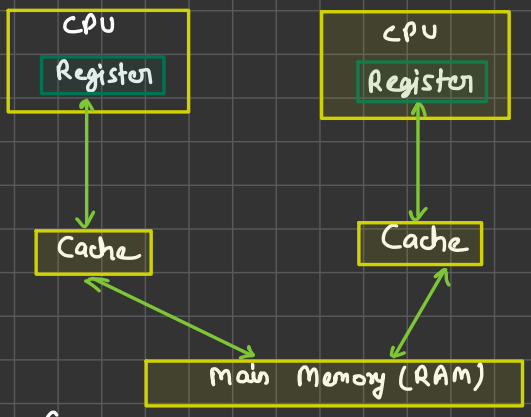
→ smallest sequence of instructions that are executed by CPU independently.
→ And 1 process can have multiple threads
→ when a process is created, it start with one thread and that initial thread known as 'main thread' and from that we can create multiple threads to perform task concurrently.

```
Thread.currentThread().getName();
```

Code Segment :-

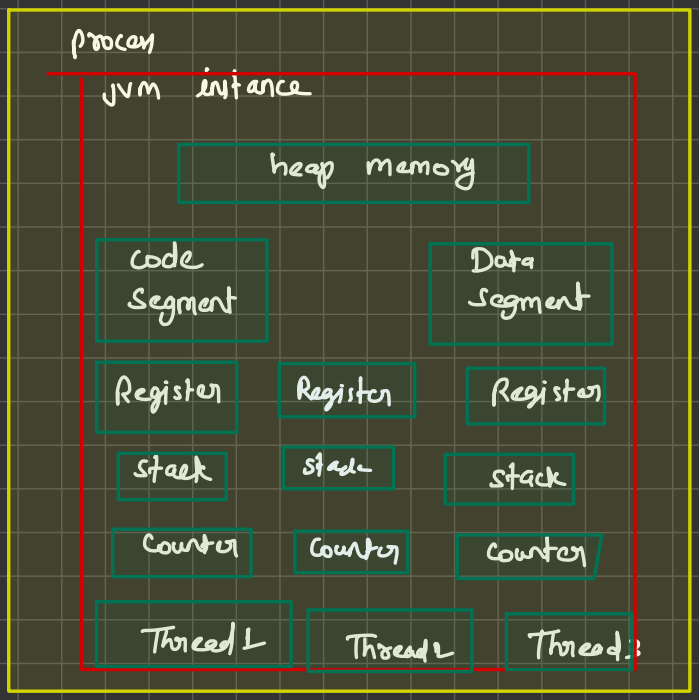
(i.e. machine code)

→ Contains the compiled BYTECODE of your 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 b/w multiple threads.



Heap :-

→ Objects created at runtime using "new" keyword are allocated in the heap.
→ Heap is shared among all threads of same process. (But not within process)
→ Threads can read and modify heap data.
→ synchronization is required b/w 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 use 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 counter after succesful execution of the instruction.

⇒ All these are managed by JVM.

→ Context switching mean CPU is giving time to every thread and seems like all threads are running simultaneously.

Definition of Multithreading:-

→ Allows a program to perform multiple task at the same time.
→ Multiple threads share the same resource such as memory space but still can perform task independently.

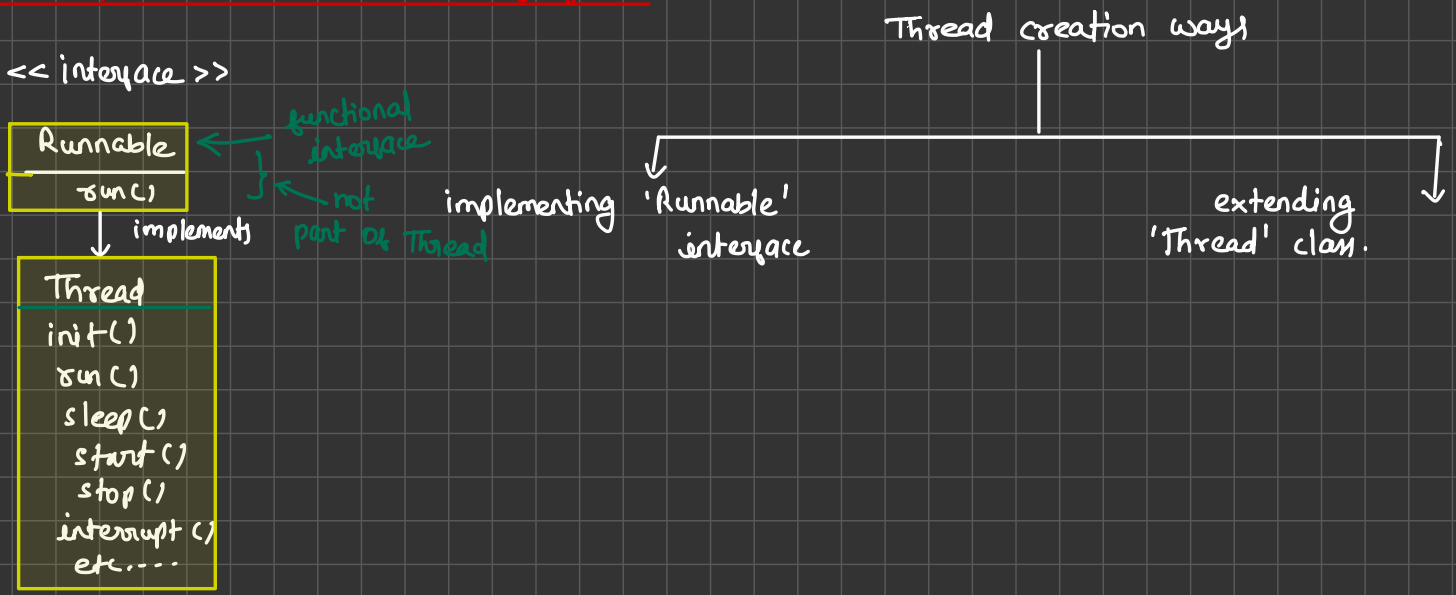
→ Benefits and challenges of Multithreading :-

Benefits:-
- improved performance by task parallelism.
- Responiveness
- Resource sharing.

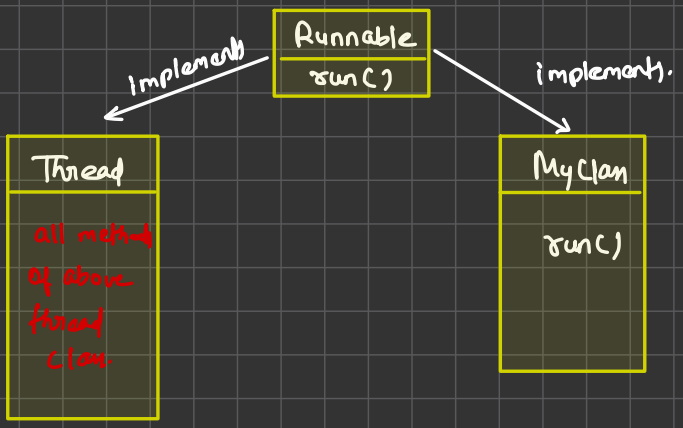
Challenges:-
- Concurrency issue like deadlock, data inconsistency etc.
- synchronized overhead.
- Testing and debugging is difficult.

→ Multitasking Vs multithreading
process, process mean MT
↳ multiple threads

⇒ Thread Creation and its LifeCycle:-



→ Implementing Runnable interface:-



step1:-

- create a class that implements 'Runnable' interface.
- implement run() method to tell the task which thread has to do.

step2 :-

- Create an instance of class that implements Runnable.
- Pass the Runnable object to the Thread constructor.
- start the thread.

```
MyClass obj = new MyClass();
Thread thread = new Thread(obj);
thread.start();
```

→ we pass the object of our class to thread class, and thread invokes the run method of this class.

2) extending 'Thread' class:-

step1:- create a Thread subclass.

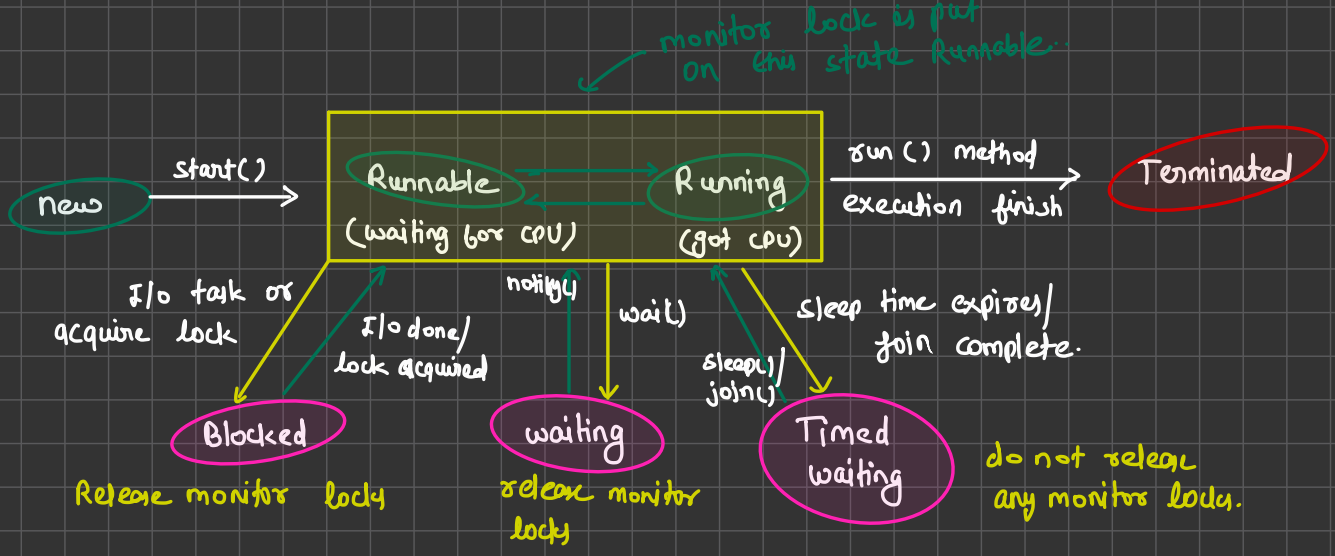
→ create a class that extends 'Thread' class.
→ override the run method to tell the task which thread has to do.

Step 2:-

- create an instance of the subclass.
- call the start() method to begin the execution.

Thread lifecycle:-

Runnable state is thread is waiting for CPU time, when CPU allocates time it moves to running state.



Life cycle state	Description
new	• Thread has been created but not started. • Its just an object in memory.
Runnable	• Thread is ready to run. • waiting for CPU time.
Running	• when thread start executing its code.
Blocked	• Different scenarios where runnable goes into the blocking state: – I/o like reading from a file or database. – Lock acquired if thread want to lock on a resource which is locked by other thread, it has to wait. • Releases all the monitor logs.
waiting	• Thread goes into this state when we call the wait() method, makes it non-runnable. • It goes back to runnable, once we call notify or notifyAll() method.
Timed waiting	• Thread waits for specific period of time and comes back to runnable state, after specific conditions met. • like sleep(), join(). • Do not release any monitor locks.

⇒ MONITOR LOCK :-

→ It helps to make sure that only 1 thread goes inside the particular section of the code(a synchronized block or method).

→ when we use synchronized method/block, so lock is put on this method/block for single thread.

Q. Implement PRODUCER CONSUMER problem.

→ Two threads, a producer and consumer, share a common, fixed size buffer as a queue.

→ The producer's job is to generate data and put it into the buffer, while the consumer's job is to consume the data from buffer.

→ The problem is to make sure that the producer won't produce data if the buffer is full, and the consumer won't consume data if the buffer is empty.

Q. why stop, Resume, Suspended methods are depricted?

STOP :- Terminates the thread abruptly, No lock release, No resource cleanup happens.

SUSPEND :- Put the thread on hold(suspend) for temporarily. No lock is released.

RESUME :- used to RESUME the execution of suspended thread.

Both this operation could lead to issue like deadlock.

Thread Joining :-

→ When thread JOIN method is invoked on a thread object. Current thread will be blocked and wait for specific thread to complete.

→ It is useful when we want to co-ordinate blw threads or to ensure we complete certain task before moving ahead.

→ if we call join on t1 that means we tell main thread to wait till t1 is done with execution.

⇒ Thread priority :-

→ priorities are integers ranging from 1 to 10.

1. → low priority

10 → highest priority

→ even we set thread priority while creation, its not guaranteed to follow any specific order, its just a hint to thread scheduler which to execute next (but its not strict rule).

→ when new thread is created, it inherit the priority of its parent thread.

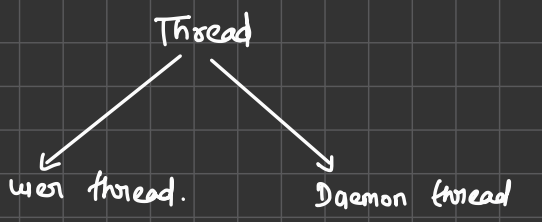
→ we can set custom priority using "setPriority(int priority)" method.

⇒ DAEMON THREAD :-

something which is running in async manner.

→ we can make a thread daemon by using

```
t1.setDaemon(true)
```

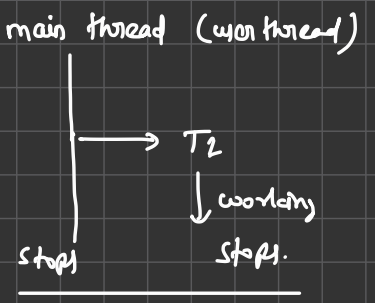


→ daemon thread is alive if any 1 user thread is alive. If all user thread finish execution, daemon thread will also stop.

→ Use :-

java garbage collector, JVM has daemon thread. In background it frees memory

→ autosave feature.
→ good for logging.



⇒ Locks and Semaphores:-

→ locking do not depend on Object like synchronize method.

(1) Reentrant lock:-

→ we can use the lock instance instead of the object.

→ so if we have a requirement that we can have different objects, but still we have to allow only one thread to go into the critical section, then instead of synchronized we have to use Reentrant lock..

```
ReentrantLock lock = new ReentrantLock();
public void producer(){
    try{
        lock.lock(); // acquire the lock
        isAvailable = true;
        Thread.sleep(4000);
    }
    catch(Exception e){
    }
    finally{
        lock.unlock();
        syso("lock relase");
    }
}
```

Intrinsic and explicit locking:-

1) **Intrinsic locking:-** also known as implicit locking and non-intrinsic locking.

Mechanism:-

→ It is achieved using the 'synchronized' keyword in java.
→ When a method or block is marked as synchronized, it acquires an intrinsic lock, also known as monitor lock.
→ Only one thread can hold this lock associated with an object at a time.

- **Usage:-** Intrinsic locking is simpler and easier to use than non-intrinsic locking. It is built into the java language and provides a straightforward way to synchronize access to shared resources.

2) **Non-Intrinsic locking:-** Also known as explicit locking.

→ **Mechanism:-**
→ It involves using explicit lock objects provided by classes like 'ReentrantLock' from 'java.util.concurrent.locks' package. These lock objects provide more flexibility and control over locking behavior compared to intrinsic locking.
→ Multiple threads can acquire the same non-intrinsic lock concurrently depending on the lock implementation and its state.

- **Usage:-** Non-intrinsic locking is typically used in scenarios where finer grained control over locking is required, such as implementing custom locking policies, supporting interruptible locks, or implementing more complex concurrency patterns.
→ Lock is acquired and released manually.

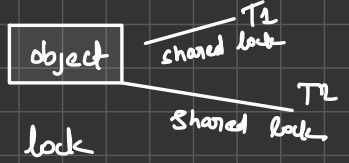
Granularity:-

Intrinsic locking operates at the level of methods or blocks while non-intrinsic locking allows finer grained locking, including the ability to lock specific sections of code or resources.

⇒ shared lock vs exclusive lock:-

→ shared lock means only read is allowed on resource.
→ exclusive lock means read/write operations are allowed.

→ If a thread gets shared lock on a resource other threads can acquire shared lock on that instance. But if shared lock is there then no thread can get exclusive lock.



→ If a thread gets exclusive lock then other threads can't get any type of lock on that resource.

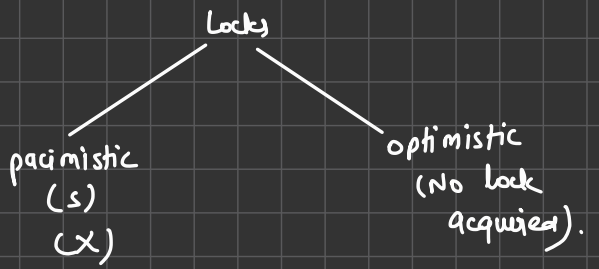
ReadWrite Lock:-

→ **ReadLock:-** more than one thread can acquire the read lock.
→ **WriteLock:-** Only one thread can acquire the write lock.

```
ReadWriteLock lock = new ReentrantReadWriteLock();

lock.readLock().lock();
lock.writeLock().lock();

- to unlock

lock.readLock().unlock();
lock.writeLock().unlock();
```

⇒ **Stamped lock:-** Read/write, Optimistic Read.

→ support Read/write lock functionality like ReadWriteLock.




```
StampedLock lock = new StampedLock();

long stamp = lock.readLock();
:
:
finally {
    lock.unlockRead(stamp);
}
```

Optimistic lock:-

→ concurrency control technique used to manage access to shared resources in a multithreaded or distributed environment.
→ It is based on the assumption that conflicts blw multiple threads accessing the same resource are infrequent

→ Instead of acquiring locks to prevent concurrent modifications optimistic locking allows multiple threads to access and modify the resource concurrently.

→ However before commiting the changes, each thread checks whether the resource has been modified by another thread since it was last accessed.
→ If no modifications have occured, the thread proceeds with its changes.
→ If modifications are detected, the thread may chose to retry its operation, merge changes, or abort and report a conflict.

→ Here is how optimistic locking typically works:-

1) **Read phase:-**
when a thread reads a resource, it also records metadata associated with the resource. such as version number or timestamp.

This metadata represents a state of the resource at the time of reading.

2) Modify phase:-

→ when the thread modifies the resource, it includes the recorded metadata with its update.
→ for e.g. it may increment the version number or update the timestamp

3) Validation phase:-

→ Before commiting the changes, the thread checks whether the resources current metadata matches the recorded metadata from the read phase.
→ If metadata matches, it means that no other thread has modified the resource since it was last read. The thread can safely commit its changes.

4) Conflict resolution:-

→ If the metadata doesn't match, it indicates that another thread has modified the resource concurrently.

→ Depending on the applications requirement the thread may chose to handle the conflict by retrying the operation; merging changes, notifying the user, or rolling back the transaction.

⇒ Semaphore Lock:-

→ It maintain a set of permits that can be acquired or released by threads.

→ In java it offers two main methods acquire() and release():-

(i) acquire:- This method is used by a thread to acquire a permit from the semaphore.
→ If permit is available, the method returns immediately; otherwise it will wait until a permit becomes available.

(ii) Release:- This method is used to release a permit back to the semaphore

→ After releasing a permit, other threads waiting to acquire a permit may be able to proceed.

```
Semaphore lock = new Semaphore( permits : 2);
try {
    lock.acquire();
    . . . .
}
finally {
    lock.release();
}
```

How many threads are allowed to

⇒ Inter thread communication :- (conditions)

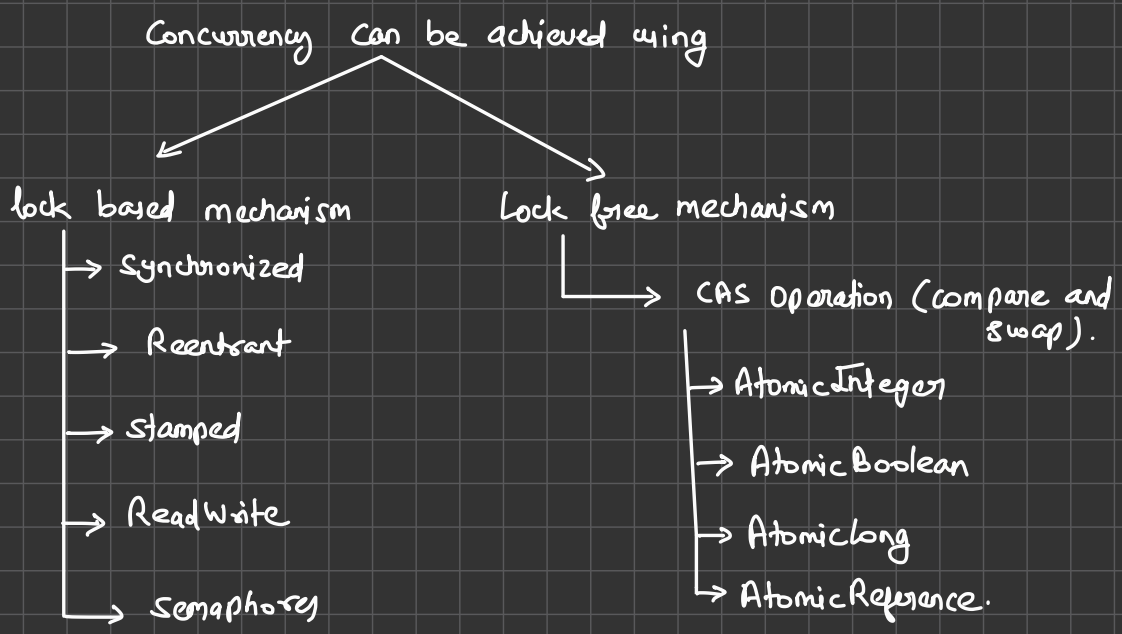
if synchronized → wait(), notify().

→ but other options available like stamped doesn't acquire monitor lock. so there comes conditions for interthread communication as wait() and notify() doesn't work here.

```
await()  = wait()
signal() = notify()
signalAll = notifyAll()
```

```
ReentrantLock lock = new ReentrantLock();
Condition con = lock.newCondition();

con.await();
    :
Condition.signal();
```

Q. In howmany ways you can achieve the Concurrency?



⇒ CAS Operation :- Compare & swap

→ Java provides these 4 classes to achieve concurrency:-

- AtomicInteger
- AtomicBoolean
- AtomicLong
- AtomicReference.

→ It uses CAS (compare and swap) technique:-

- It's a low level operation. (supported by CPU).
- It's atomic. (single unit operation, no matter the number of cores in CPU).
- And all modern processor supports it.

→ 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.

→ Optimistic locking works in DB and in java cpu has this CAS operation.

e.g. ABA problem :- lets say 1 thread went to memory and other thread t2 changed value to 12 and t3 changed it back to 10, and then t2 comes to change it read value was 10, but it's not the 10 that was read by t1 before.

CAS(m1, 10, 12) — 10 = expected, 12 = new value

→ This can easily be resolved by adding version number or timestamp.

⇒ Atomic variables :-

What does atomic mean?
→ It means single or "all or nothing" like transaction.

```
class SharedResource {
    int counter;
    public void increment(){
        counter++;
    }
    public int get(){
        return counter;
    }
}
```

```
SharedResource resource = new SharedResource();
for(int i=0; i<400; i++){
    resource.increment();
}
sysO(resource.get());
```

3 operations need to be done

→ if we use 2 threads and run on shared resource we will get a different value for counter than what is expected.

① load counter value.
② increment it by 1.
③ Assign back.

} not atomic operation so not thread safe.

→ There are two solutions we know of :-
① Using lock like synchronized.
② Using lock free operation like AtomicInteger.

```
AtomicInteger counter = new AtomicInteger(0);
```

```
counter.incrementAndGet()
```

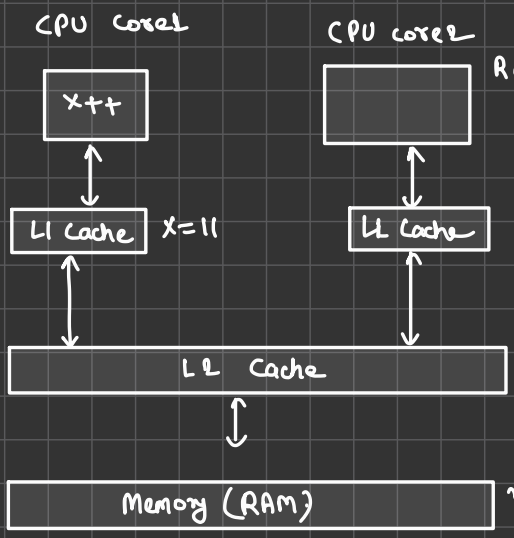
Volatile values directly gets from memory not from cache.

```
do {
    expected read value variable
} while( !CAS( mem, offset, expected, x);
```

↳ keep trying till satisfied condition is met.

Read, modify, update in that scenario go for atomic, otherwise go for synchronized.

Atomic	Volatile
- thread safe	- No relation with thread safe.



Read x → check in L2 first then to memory (IO).

→ but in core1 it has incremented it to 11 but memory is not yet updated, value is in cache.

→ If we set a variable to volatile then read and write directly happens to memory and not cache.

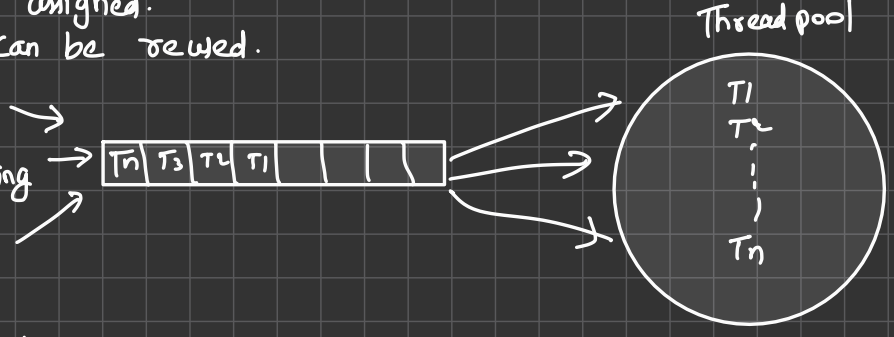
Collection	Concurrent Collection	Lock
PriorityQueue	PriorityBlockingQueue	ReentrantLock
LinkedList ArrayDeque	ConcurrentLinkedDeque	Compare and Swap operation.
ArrayList	CopyOnWriteArrayList	ReentrantLock
HashSet	newKeySet method inside ConcurrentHashMap	synchronized
TreeSet	Collections.synchronizedSortedSet()	synchronized.
LinkedHashSet	Collections.synchronizedSet	synchronized
Queue Interface	ConcurrentLinkedQueue	Compare and Swap

Q. What is Thread Pool?

→ It's a collection of threads (aka workers), which are available to perform the submitted task.

→ Once task completed, worker thread get back to thread pool and wait for new task to be assigned.

→ Means threads can be reused.



Why thread pool needed?

→ thread creation takes time.

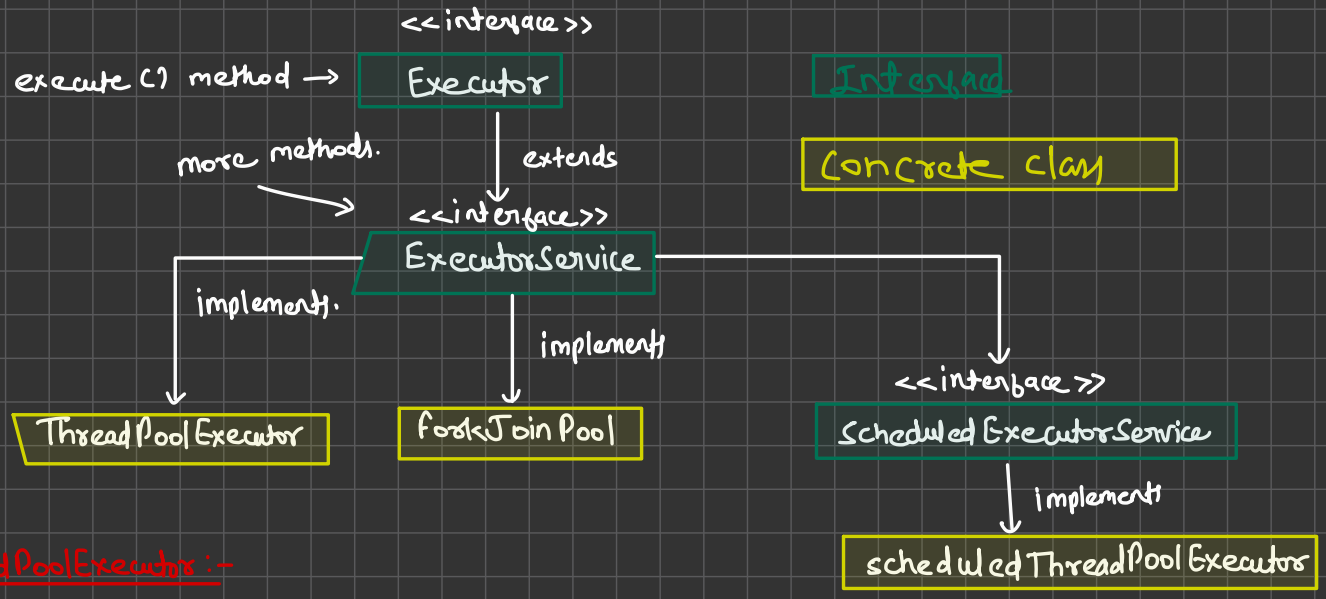
→ need to manage thread throughout its lifecycle.

→ More threads means, more context switching time, using control over thread creation, excess context switching can be avoided.

⇒ Executor framework :-

The executor framework in java provides a standardized way of executing tasks concurrently in a multithreaded environment.

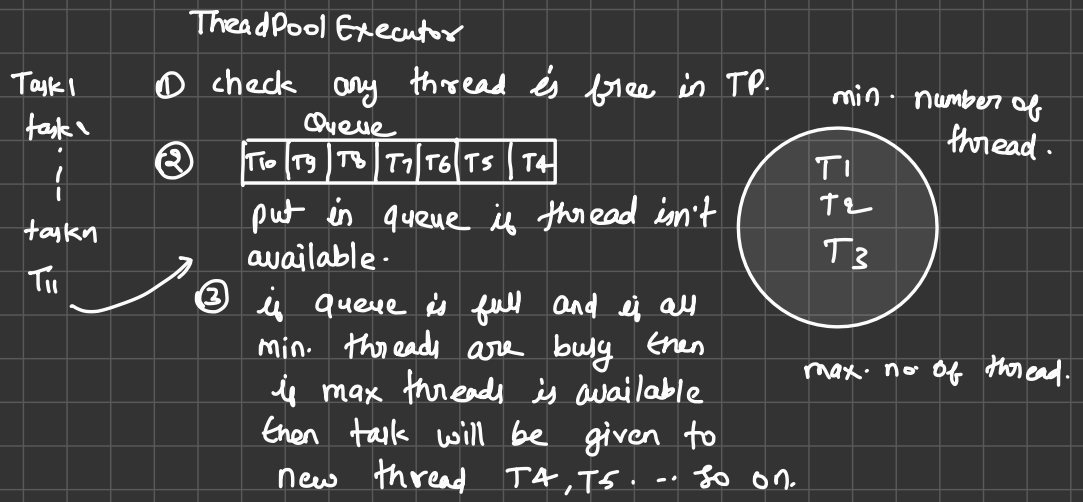
→ It abstracts away the complexity of managing threads manually and allows developers to focus on writing tasks while leaving the details of thread management to the framework.



→ ThreadPoolExecutor :-

→ It helps to create a customizable ThreadPool.

process flow :-



→ ThreadPoolExecutor constructor:-

```
public ThreadPoolExecutor(int corePoolSize, int maxPoolSize,
        long keepAliveTime, TimeUnit unit,
        BlockingQueue<Runnable> workQueue,
        ThreadFactory factory, RejectedExecutionHandler handler)
```

→ It has these parameters:-

→ corePoolSize:-

Number of threads initially created and kept in the pool, even if they are idle.

→ allowCoreThreadTimeout:-

→ If this property is set to true (by default its false), idle thread kept alive till time specified by 'keepAliveTime'.

→ keepAliveTime:-

→ Thread which are idle get terminated after this time.

→ MaxPoolSize:-

→ Maxm number of thread allowed in a pool.

→ if no of thread == corePoolSize and queue is also full, then new threads are created. (till its less than 'maxPoolSize').

→ Excess thread, will remain in pool, this pool is not shutdown or if allowCoreThreadTimeOut set to true, then excess thread get terminated after remain idle for keepAliveTime.

→ TimeUnit:-

→ TimeUnit is for keepAliveTime, whether Millisecond or Second or Hours etc.

→ BlockingQueue:-

→ Queue used to hold task, before they got picked by the worker thread.

BoundedQueue:- Queue with FIXED capacity.

→ like: ArrayBlockingQueue.

UnBoundedQueue:-

→ like: LinkedBlockingQueue.

ThreadFactory:-

→ Factory for creating new thread. ThreadPoolExecutor use this to create new thread, this factory provide us an interface to:

- Give custom thread name.
- To give custom thread priority.
- To set Thread Daemon flag etc.

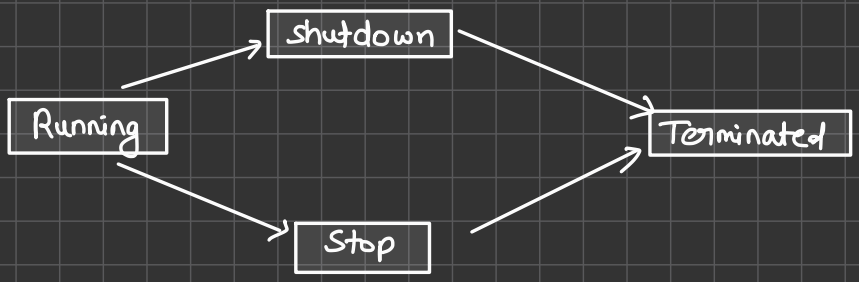
→ RejectedExecutionHandler:-

→ Handler for task that can't be accepted by thread pool.

→ Generally logging logic can be put here. For debugging purpose:

- new ThreadPoolExecutor.AbortPolicy
  - throws RejectedExecutionException.
- new ThreadPoolExecutor.CallerRunsPolicy
  - Executes the rejected task in caller Thread (thread that attempted to submit the task.
- new ThreadPool/Executor.DiscardPolicy
  - Silently discard the rejected task, without throwing any exception.
- new ThreadPoolExecutor.DiscardOldestPolicy
  - → Discard the oldest task in the queue, to accomodate new task.

⇒ Lifecycle of ThreadPoolExecutor :-



→ **Running.-**
- Executor is in running state and submit method will be used to add new task.

→ **shutdown :-**
- Executor do not accept new tasks, but continues to process existing tasks, once existing tasks finished, executor moves to terminate state.
  - → Method used shutdown().

→ **Stop (force shutdown):-**
- executor do not accept new tasks.
- executor forcefully stops all the tasks which are currently executing.
- And once fully shutdown, move to terminate state.
- method used shutdownNow().

→ **Terminated:-**
- → End of life for particular ThreadPoolExecutor.

→ isTerminated() method can be used to check if particular thread pool executor is terminated or not.

Q. Custom thread factor and how to use it?

→ you can control how threads are created for thread pool executor.

```
class CustomThreadFactory implements ThreadFactory{

    @Override
    public Thread newThread(Runnable r){
        Thread th = new Thread(r);
        th.setPriority(Thread.NORM_PRIORITY);
        th.setDaemon(false);
        return th;
    }
}
```

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

Soln 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.

→ And its an iterative process to update the min and max values based on monitoring.
→ formula to find the no. of threads:-

Max No of thread = No. of CPU core * (1+ Request waiting time/process time)

max no of active tasks = task arrival rate * task execution time.

→ But this formula, do not consider memory yet, which need to be considered.

per thread space = 5MB * no. of threads (includes thread stack space)

Q. Future, Callable and CompletableFuture:-

→ what if caller want to know the status of thread. whether its completed or failed.etc.

→ whenever we call submit() its return type is future.

→ **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.

```
Future<?> futureObj = poolExecutor.submit(() -> sysout(""));
    Syso( futureObj.isDone());
```

S.NO	Methods Available in Future Interface	Purpose
1.	boolean cancel(boolean mayInterruptIfRunning)	• Attempts to cancel the execution of task. • Returns false, if task can't be cancelled (typically bcoz task already completed); returns true otherwise.
2.	boolean isCancelled()	Returns true, if task was cancelled before it got completed.
3.	boolean isDone()	• Returns true if this task completed • Completion may be due to normal termination an exception or cancellation -- in all of these cases, this method will return true.
4.	V get()	• wait if required, for the completion of the task. • After task is completed, retrieve the result if available. • will wait for infinite time for task completion.
5.	V get(long time, TimeUnit unit)	• wait if required for atmost given timeout period. • Throws 'TimeoutException' if timeout period finished and task is not yet complete.

⇒ **Callable:-** submit(Runnable), submit(Runnable, T), submit(Callable<>).

? → can be anything.

→ In the executor framework of java Callable is similar to a 'Runnable', but it can return a result and throw a checked exception.
→ It is a functional interface that represents a task that can be executed concurrently and returns a result.

1) Defining a task:-

```
public class MyTask implements Callable
                                <String>{

    @Override
    public String call() throws Exception{
        //
        return "Task Completed");
    }
```

```
@FunctionalInterface
public interface Callable<V>{
    V call() throws Exception;
}
```

```
Callable<String> task = new MyTask();

Future<String> future = executor.submit(task);
```

⇒ Completable future :-

→ Introduced in java8.
→ To help in async programming.
→ we can consider it as an advanced version of future provides additional capability like chaining.

→ 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 executor)
```

→ supplyAsync method initiates an Async operation.
→ 'supplier' is executed asynchronously in a seperate thread.
→ If we want more control on threads, we can pass Executor in the method.
→ By default it uses, shared ForkJoin pool executor. It dynamically adjust its pool size based on processors.

2) thenApply & thenApplyAsync:-

→ Apply a function to the result of previous Async computation.
→ Return a new CompletableFuture object.

```
CompletableFuture<String> asyncTask1 = CompletableFuture.supplyAsync(() ->{
                                            return "Hello";
                                        }, poolExecutor).thenApply((String val) ->{
                                            return val + "world";
                                        });
```

→ thenApply method:-

→ Its synchronous execution.
→ Means, it use same thread which completed the previous async task.

→ thenApplyAsync method :-

- Its asynchronous execution
- Means it use different thread (from fork-join pool, if we don't provide the executor in the) complete this function.
- If multiple 'thenApplyAsync' is used, ordering can't be guarantee, they will run concurrently.

→ thenCompose and thenComposeAsync :-

- chain together dependent Async operations.
- Means when next async operation depends on the result of the previous async one we can tie them together.
- for async tasks, we can bring some ordering using this.

→ thenAccept() and thenAcceptAsync :-

- Generally end stage, in the chain of Async operations.
- It does not return anything.

→ thenCombine and thenCombineAsync :-

- used to combine the result of 2 comparable future.

⇒ ThreadPool : Executors utility class and Fork-Join Pool :-

→ Executors provide factory methods which we can use to create Thread Pool Executor.
→ Present in "java.util.concurrent" package.

1) FixedThreadPoolExecutor :-
→ '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.

```
ExecutorService poolExecutor = Executors.newFixedThreadPool(nThread:5);
poolExecutor.submit(() -> "this is the async task");
```

⇒ Cached ThreadPool Executor :-

→ 'newCachedThreadPool' method creates a thread pool that creates a new thread as Needed (dynamically).

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

3) Single Thread Executor :-

→ 'newSingleThreadExecutor' creates executor with just single worker thread.

Min Max	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.

⇒ Work Stealing Pool Executor :-

→ It creates a Fork-join Pool Executor.

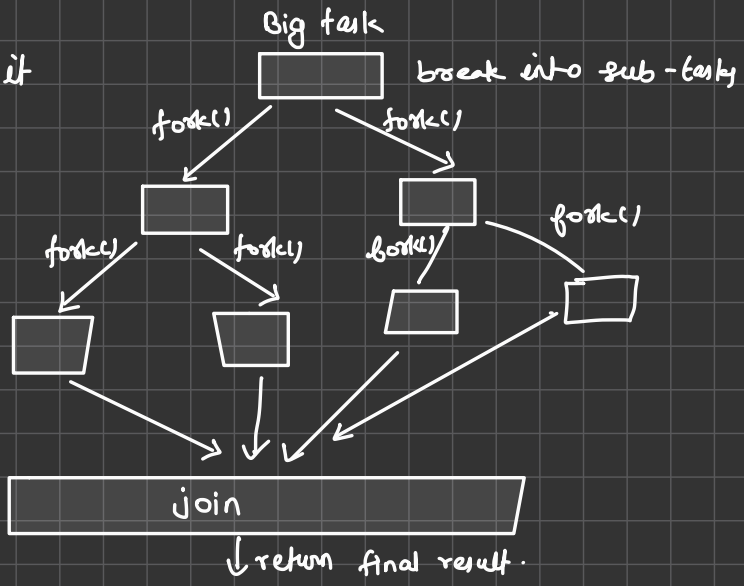
→ Number of threads depends upon the Available processors or we can specify in the parameter.

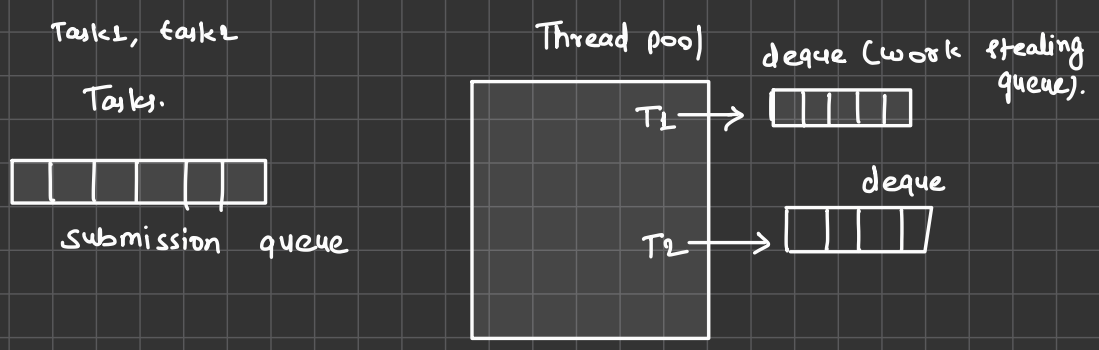
→ One big task, only 1 thread can access it at a time.

→ If we divide it to multiple subtasks then different threads can work on each subtask

→ so this functionality is provided by fork-join pool.

→ so this way we bring more parallelism





→ work stealing queue is associated with each thread.

→ Thread will start working on on one subtask, and rest will be put int work stealing queue associated with that thread.

priorities:-

(i) check if any task is there in work stealing queue.
(ii) check if submission queue has any task.
(iii) Can thread steal some task from busy thread, if so steal it and put in it's own stealing queue and start working on subtask.

How to divide the task into subtasks & and how to join

→ Tasks can be split into multiple small tasks. For that task should extend :-
- RecursiveTask → when subtask returns a value.
- RecursiveAction. → when subtask doesn't return a value.

→ we can create fork-join pool using "newWorkStealingPool" method in ExecutorService.
or
By calling ForkJoinPool.commonPool() method.

⇒ Shutdown vs awaitTermination 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:-

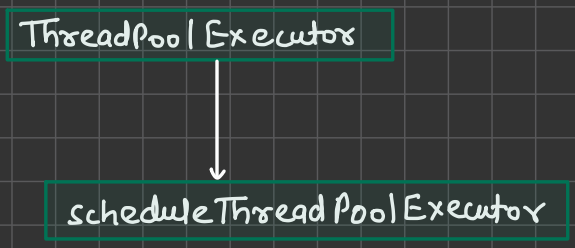
→ It's an optional functionality. Return true/false.
→ It is used after calling 'shutdown' method.
→ Blocks calling thread for specific timeout period, and wait for ExecutorService shutdown.
→ Return true, if ExecutorService gets shutdown within specific timeout

else false.

⇒ **shutDownNow :-**

- Best effort attempt to stop/interrupt the actively executing tasks.
- Halt the process of tasks which are waiting.
- Return the list of tasks which are awaiting execution.

⇒ **ScheduledThreadPoolExecutor :-** Helps to schedule the tasks.



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).	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 completed, new task can't be started (it will wait in Queue).

⇒ **ThreadLocal :-**

→ ThreadLocal class provide access to Thread-local variables.
→ This 'Thread-Local' variable hold the value for particular thread.
→ Means each thread has its own copy of thread-local variable.
→ We need only 1 object of ThreadLocal class and each thread can use it to set and get its own Thread-variable variable.

⇒ Virtual Thread vs Platform Thread:-

→ Moto of virtual thread:-
- To get higher throughput not latency.

```
Thread thL = Thread.ofVirtual().start(RunnableTask);
                    or
ExecutorService myExecutorObj = Executors.newVirtualThreadPerTaskExecutor();
myExecutorObj.submit(RunnableTask);
```

Normal Thread
(platform thread)

→ when t1.start() is executed JVM will tell to OS to create one native thread.
→ so if you are creating 10 threads then 10 OS threads will be created by OS.

☆☆
→ So JVM just provides wrapper for OS threads as platform threads.

→ Disadvantage:-

→ its slow, thats why we we ThreadPoolExecutor. t1.start() is system call.
→

⇒ Virtual Thread:-

→ JVM can create any number of threads, and these threads are totally managed by JVM.

→ so if one virtual thread want the CPU JVM attaches it to the OS thread and when it goes to waiting state JVM detaches it and attaches another thread that needs CPU.