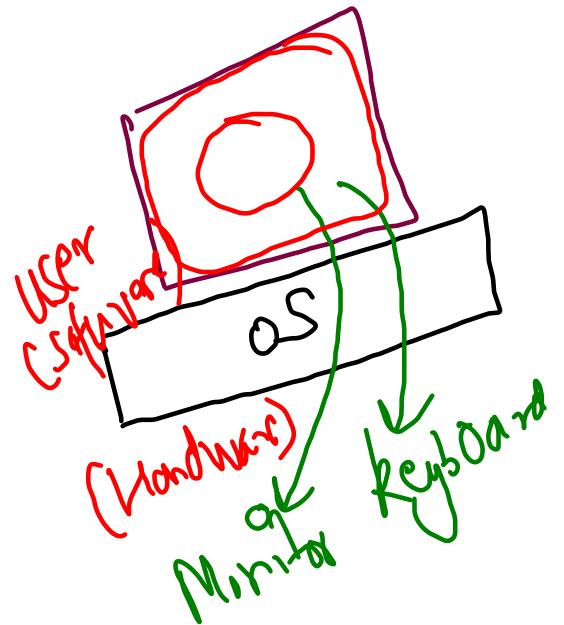


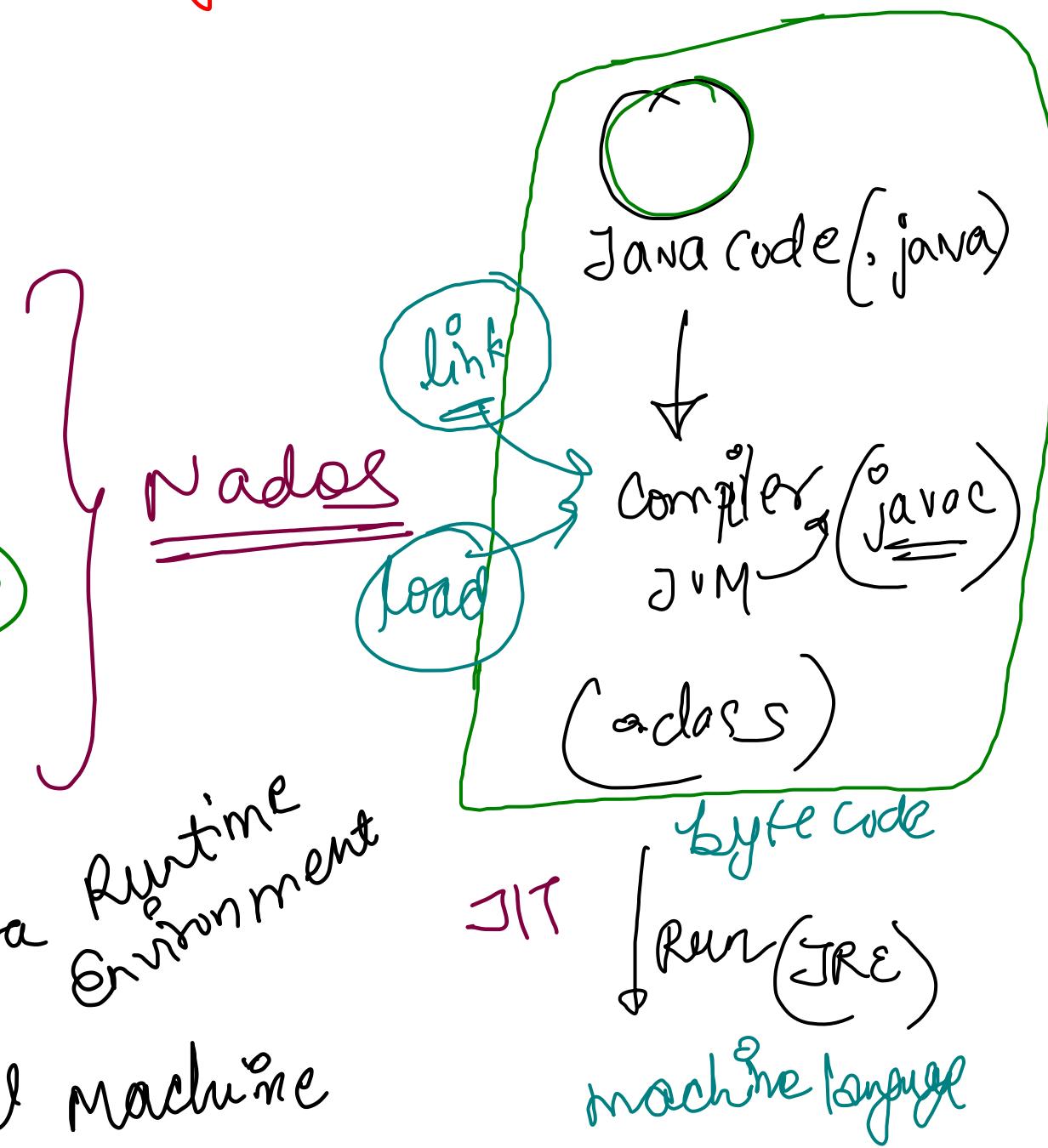
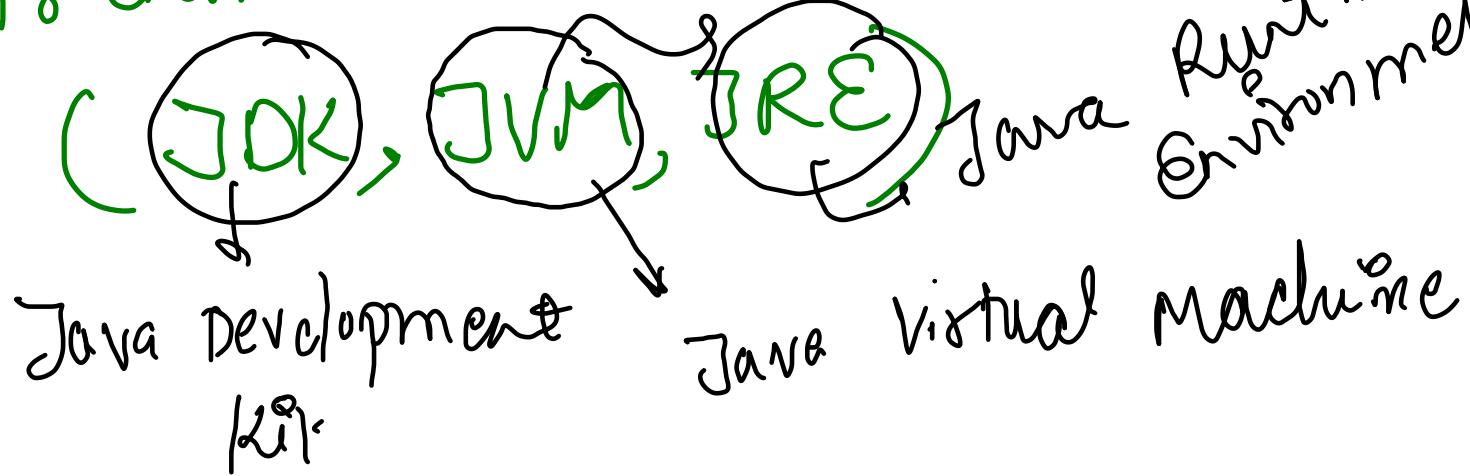
Object Oriented Programming in Java



Java Basics

- Flow of a program
- Why Java? (Multiplatform)

Architecture Java



Pillars of Object oriented Programming

Encapsulation

- Classes & Objects
- Constructors
- { Overloading & Chaining }
- this keyword
- static keyword
- Getters & Setters

Inheritance

- ISA vs has A
- Types of Inheritance
- super keyword
- Multiple Inheritance
{ Diamond Problem & Workaround }

Poly morphism

- Method Overloading
{ Compile-time }
- method overriding
{ Run-time }

Abstraction

- & Data Hiding
- Access Modifiers
- Packages
- Abstract class
- Interface

Object Oriented Programming

INTRODUCTION

→ **Classes** vs **Objects**

→ logical entity { blueprint of a object }

→ physical entity { occupies space }
 { instance of a class }

→ Data Members / Instance variables
 → properties of objects

→ Member functions / methods
 → behavior of objects

→ Getters & Setters

→ g; up

Integer. MAX-VALUE
Integer. MIN-VALUE
Integer.parseInt()

Dynamic Memory Allocation

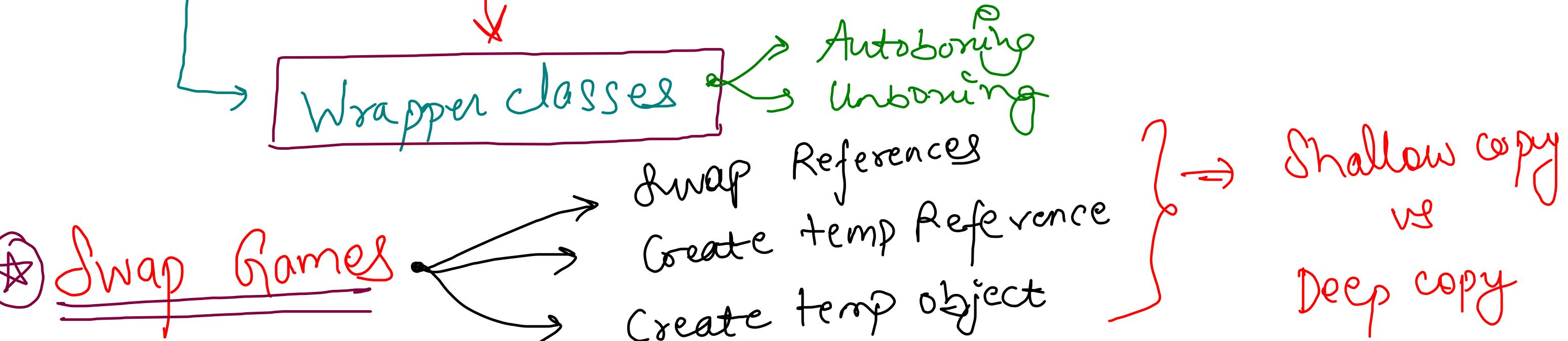
→ Memory mapping { Reference Variables in stack vs Object in heap }

int → Integer
long → Long

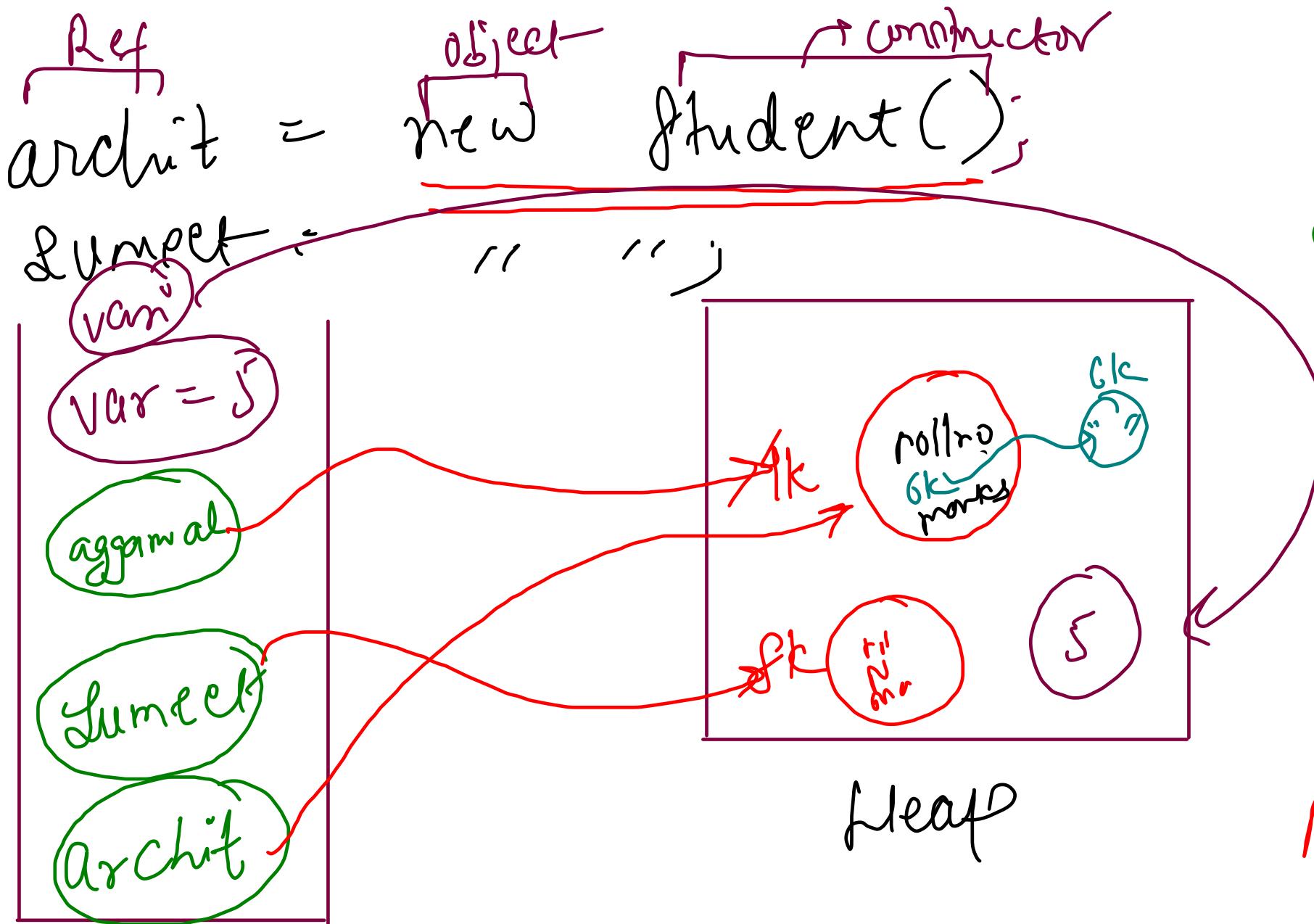
Primitive data types → Stack vs Objects.

Is Java a pure object oriented programming language?

NOT



class
Student
{
 ...
}



Stack

```
int var = 5; new Integer(5)
Integer Var = Var;
System.out.println(Var);
```

Student
aggarwal
= archit;

~~Memory Mapping~~

```

public static void swap3(Student a, Student b){
    Student temp = a;
    a.marks = b.marks;
    a.rollNo = b.rollNo;
    b.marks = temp.marks;
    b.rollNo = temp.rollNo;
}

```

half
deep
change

```

public static void swap2(Student a, Student b){
    Student temp = new Student();
    temp.marks = a.marks;
    temp.rollNo = a.rollNo;

    a = b;
    b = temp;
}

```

local member
no change

```

public static void swap1(Student a, Student b){
    Student temp = a;
    a = b;
    b = temp;
}

```

local member
no change

```

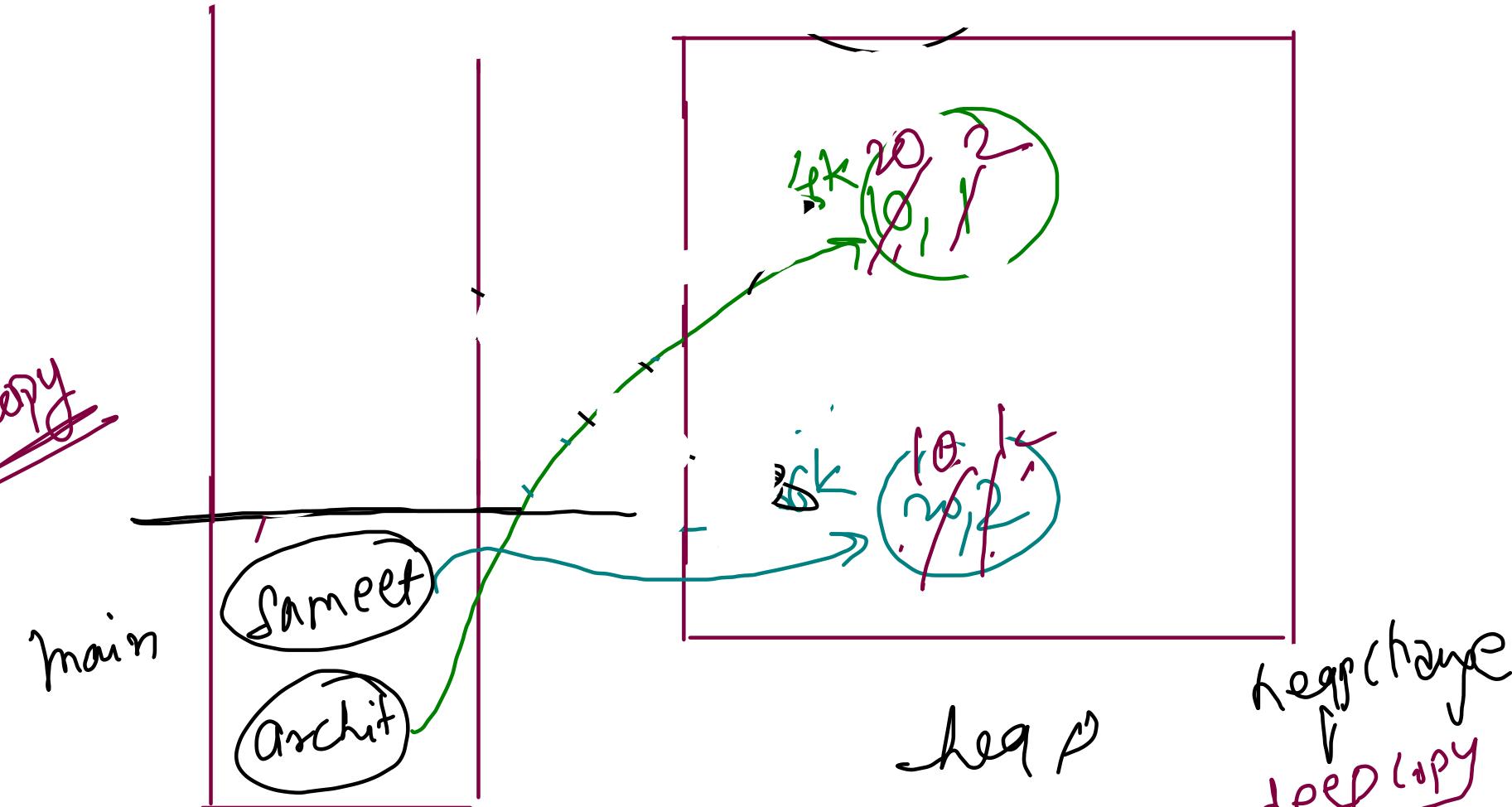
public static void main(String[] args){

    Student archit = new Student();
    archit.marks = 10;
    archit.name = "Archit";
    archit.rollNo = 1;

    Student sumeet = new Student();
    sumeet.marks = 20;
    sumeet.name = "Sumeet";
    sumeet.rollNo = 2;

    System.out.println(archit.marks + " " + archit.rollNo);
    System.out.println(sumeet.marks + " " + sumeet.rollNo);
    swap(archit, sumeet);
    System.out.println(archit.marks + " " + archit.rollNo);
    System.out.println(sumeet.marks + " " + sumeet.rollNo);
}

```



Stack
A . D
swap1,2 swap3

```

public static void swap4(Student a, Student b){
    Student temp = new Student();
    temp.marks = a.marks;
    temp.rollNo = a.rollNo;

    a.marks = b.marks;
    a.rollNo = b.rollNo;

    b.marks = temp.marks;
    b.rollNo = temp.rollNo;
}

```

10, 1
20, 2 B
20, 2
10, 1

→ Constructor

- Default Constructor (Implicit)
- Default Constructor (Explicit)
- Parameterized Constructors
- Copy Constructor
 - Shallow Copy
 - Deep copy

Constructor Overloading

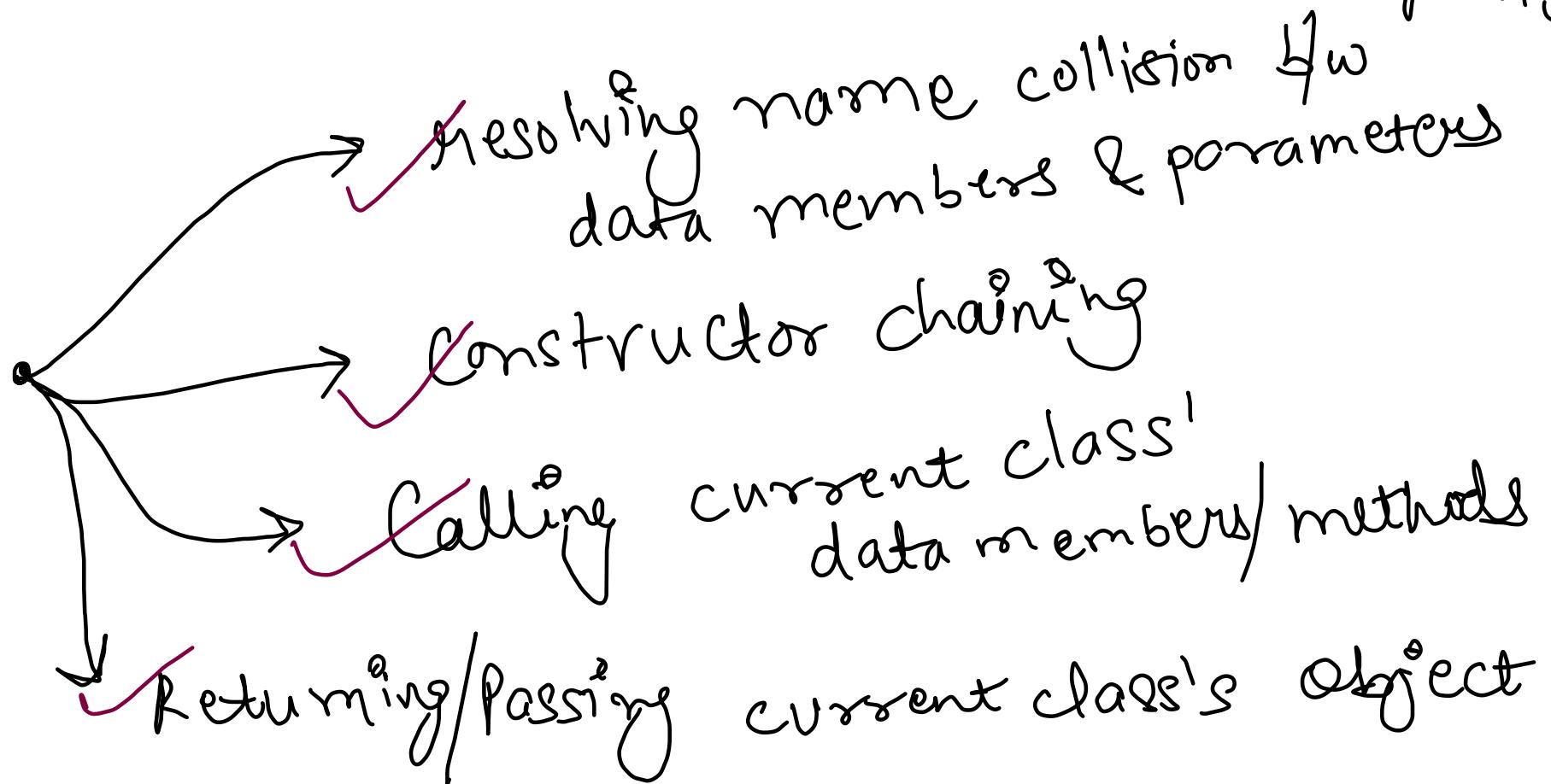
- No arguments
- Order of arguments
- Upcasting

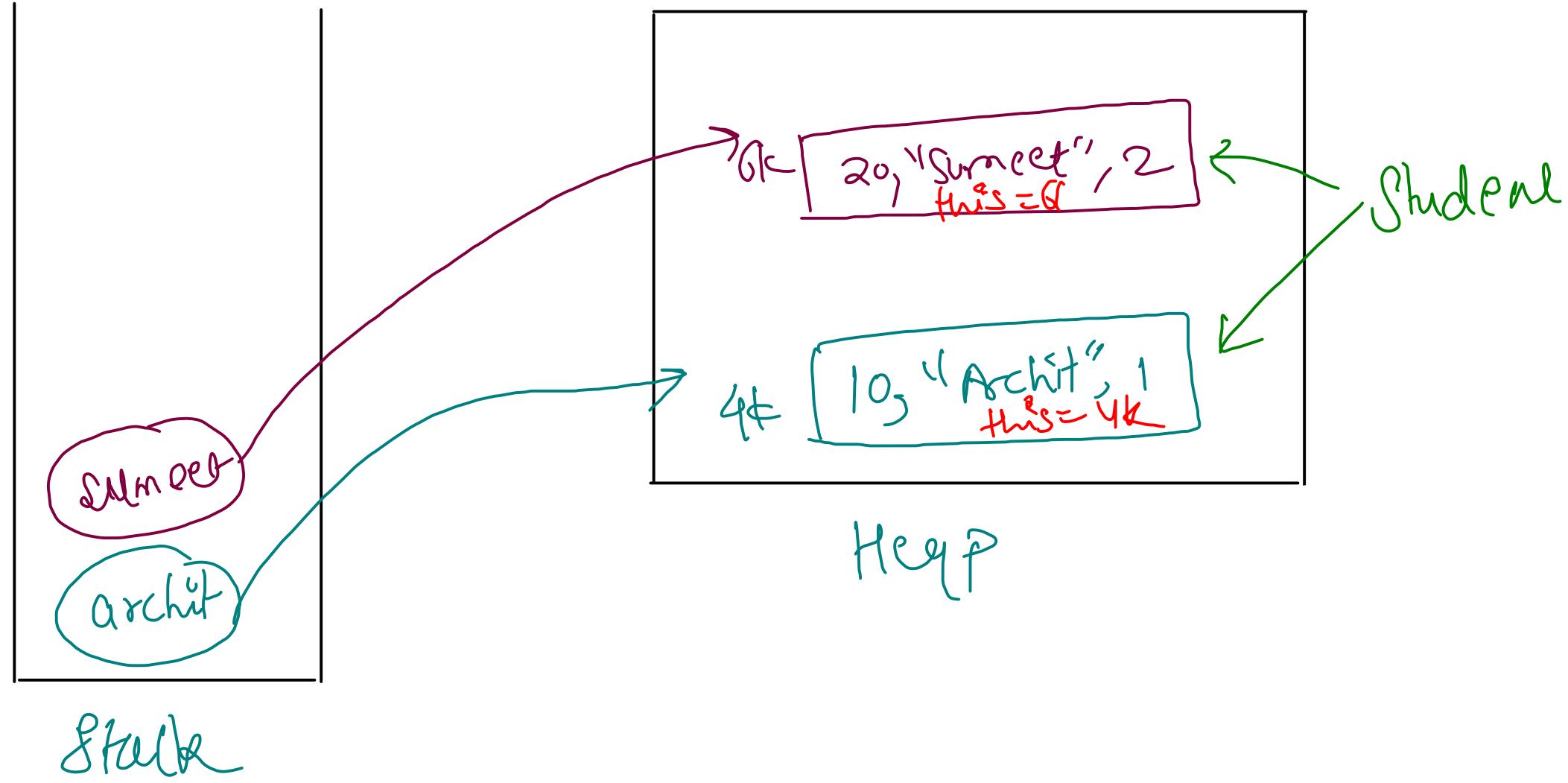


THIS

KEYWORD

↓
Self-referential
pointer



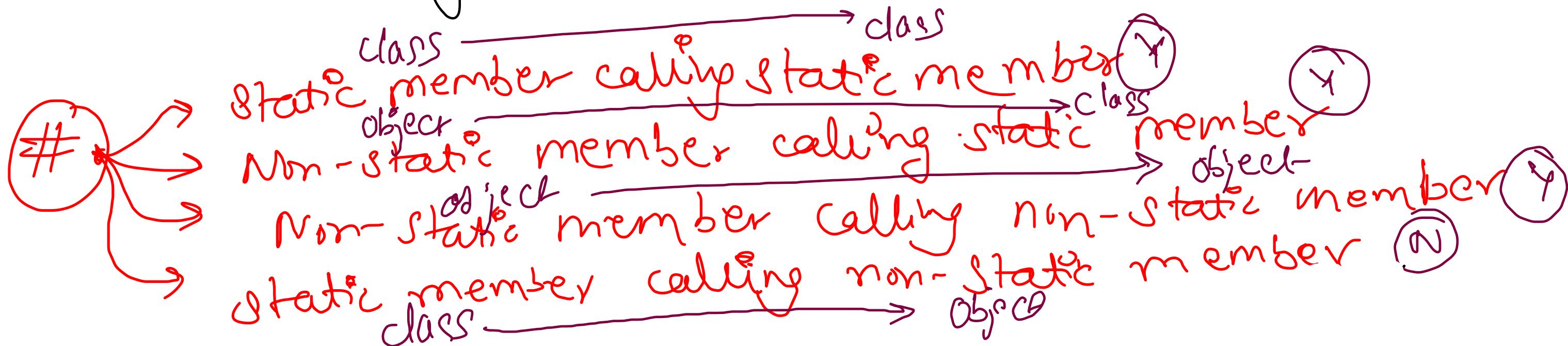


STATIC Keyword

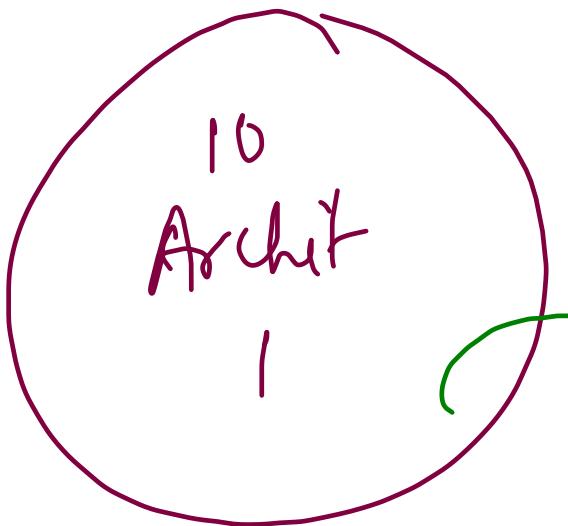
- ① Data Members → Instance variables vs class variables
Accessing static members using class name
- ② Member Functions/ Methods

→ Why main() is static? Main class → instance
(main()) → static

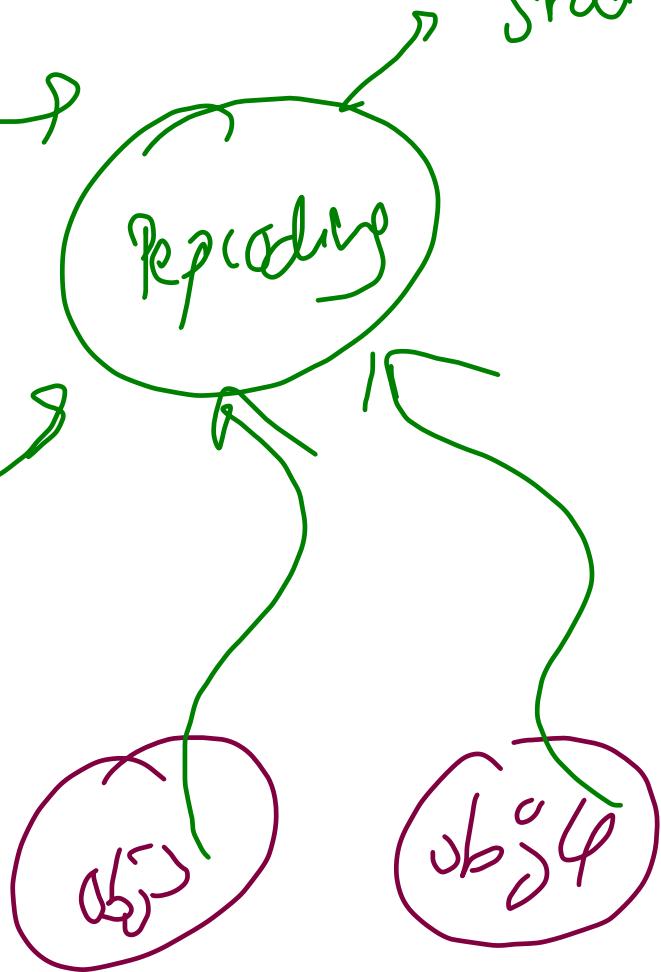
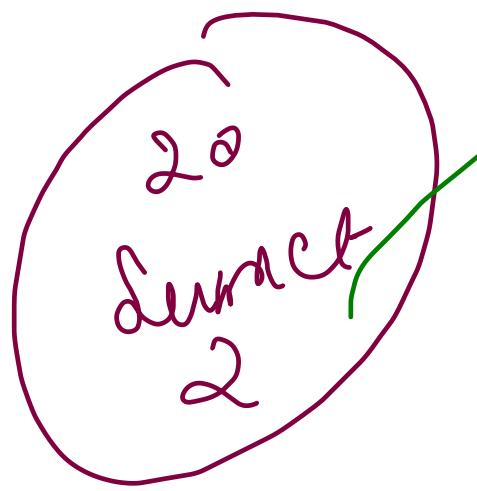
→ this keyword in static method?



Archit



sumact



static property
(class property)

~~③~~ Nested class \rightarrow {Create objects of inner class
w/o instantiating outer
class}

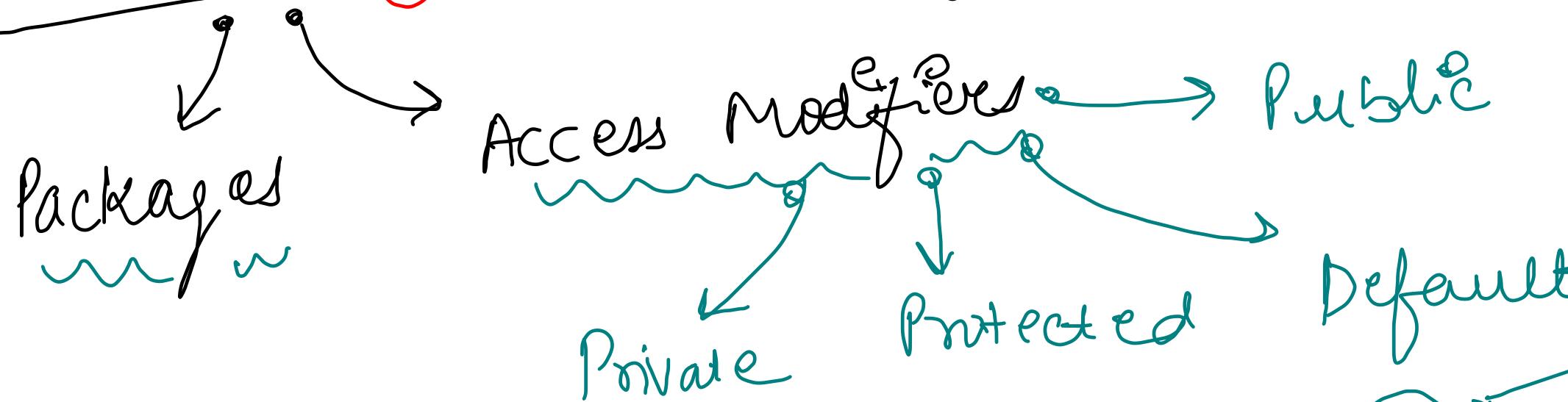
static block

It runs only once during the creation
of first object of class

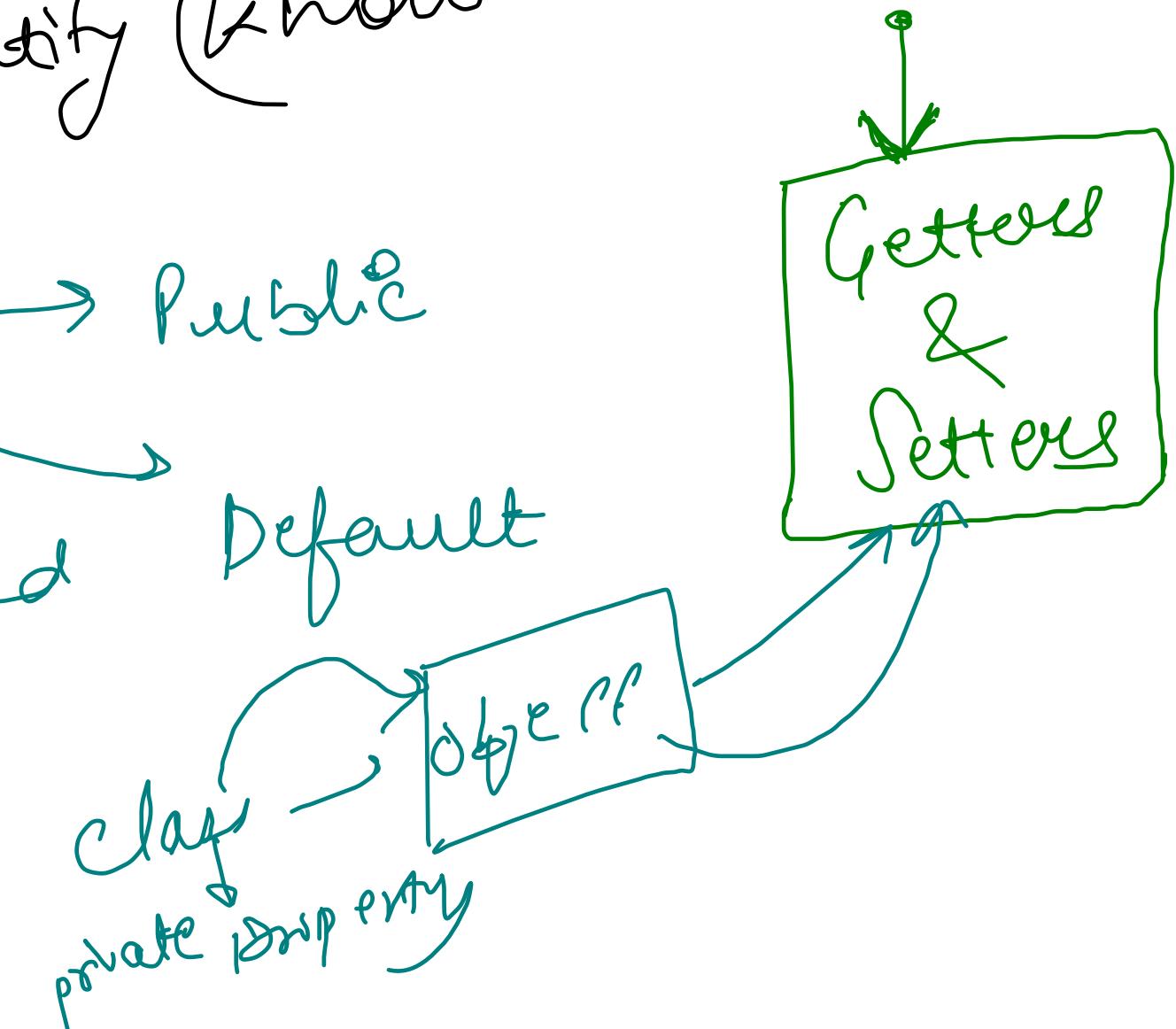
It runs only when class is loaded
onto the RAM.

Encapsulation

Implement
Data Hiding

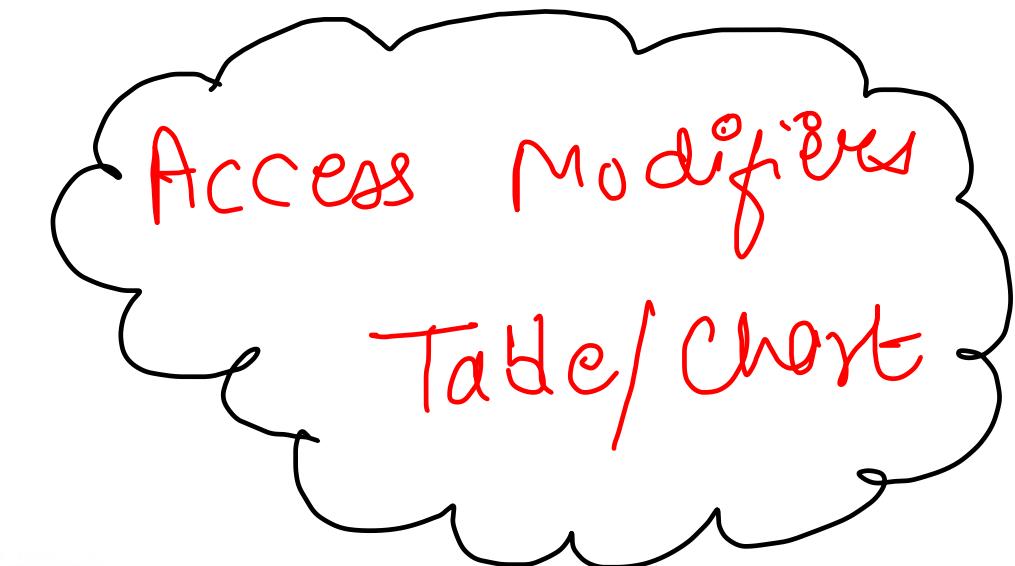


"Wrapping together the data members and member functions into a single entity (known as class)"



Let us see which all members of Java can be assigned with the access modifiers:

Members of JAVA	Private	Default	Protected	Public
Class	No	Yes	No	Yes
Variable	Yes	Yes	Yes	Yes
Method	Yes	Yes	Yes	Yes
Constructor	Yes	Yes	Yes	Yes
interface	No	Yes	No	Yes



Most Restrictive ← → Least Restrictive

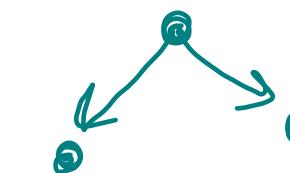
Access Modifiers ->	private	Default/no-access	protected	public
Inside class	Y	Y	Y	Y
Same Package Class	N	Y	Y	Y
Same Package Sub-Class	N	Y	Y	Y
Other Package Class	N	N	N	Y
Other Package Sub-Class	N	N	Y	Y

Inheritance

Types of Inheritance

- Single Inheritance
- Multi-level Inheritance
- Hierarchical Inheritance

Parent child → child class
superclass → subclass



SUPER keyword

refers to immediate parent class' object

Constructor chaining in Inheritance
access data members/ methods of parent class.

Multiple Inheritance

→ Why not possible in Java?

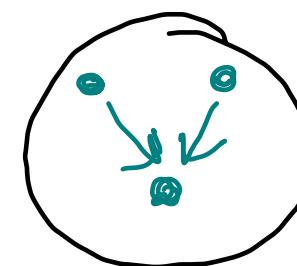
Object class

{ Superclass of
all classes }

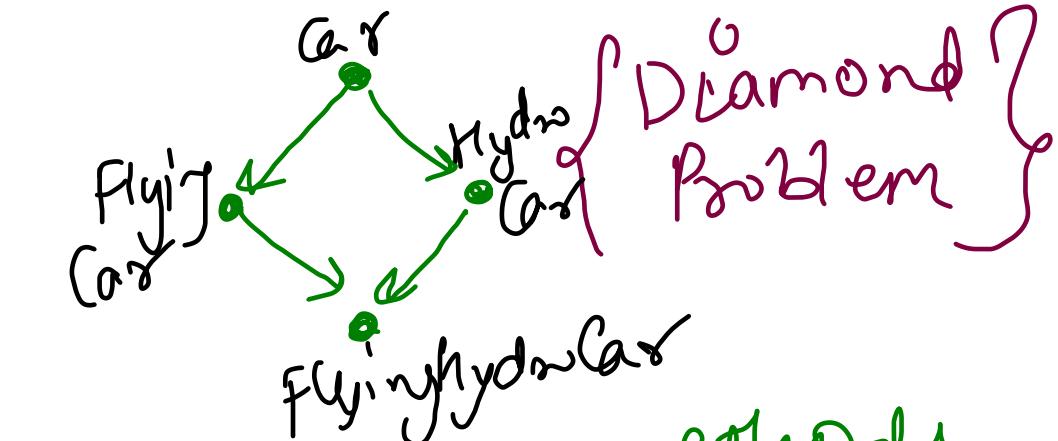
→ What type of Inheritance?
Multilevel

→ methods in object class

{eg equal, toString, hashCode,
finalize, clone}



Hybrid Inheritance



Parent class methods
Collision resolution?

flying, Hydro
↳ search cycle

Solution

→ multiple interfaces can be implemented by a single class.

is A relationship

{Inheritance}

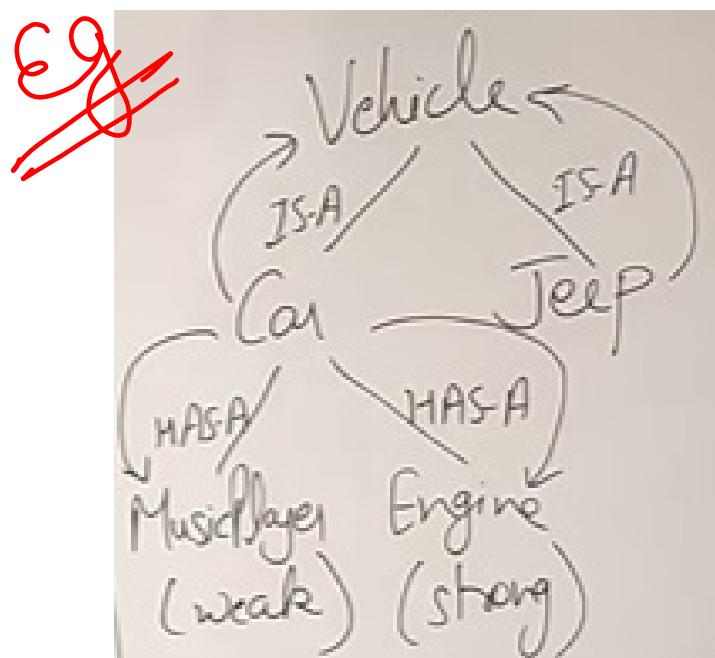
VS

has A relationship

{Association}

→ Tightly Coupled Relationship

Direct access through Super keyword



→ Loosely coupled Relationship

Accessing using Object

Example → Nested class
→ Object Member

Type of Association (has A)

- Aggregation (Weak)
- Composition (Strong)

Polymorphism → One action in many ways/forms

single class
2 functions

Types of Polymorphism

Compile-time polymorphism

or
Static polymorphism

{ Method Overloading }

- Number of arguments ✓
- Order of arguments ✓
- Return type ✗

Automatic type conversion

Run-time polymorphism
or
Dynamic Polymorphism

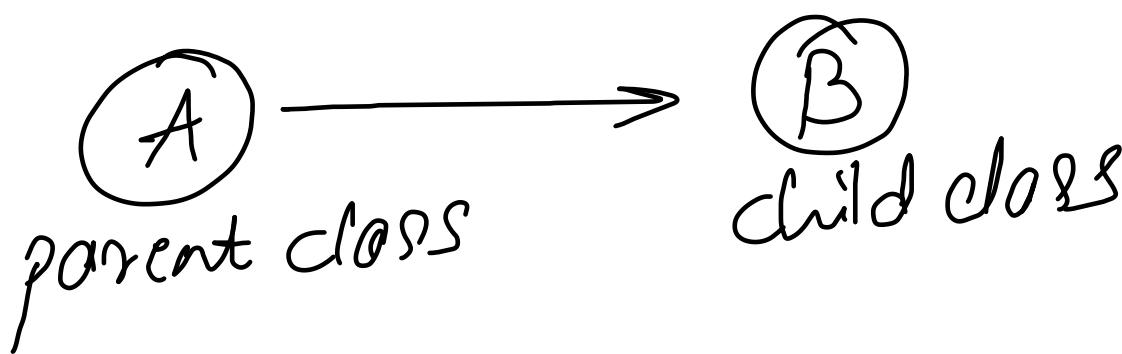
or { Dynamic Method Dispatch }

{ Method overriding } ↗ Inheritance
Parent Ref
child obj

- Same
 - No of arguments
 - Type of arguments
 - Return type

ClassName → gives the functions which we can call
referenceName = new ClassName()

→ provides the implementation of those funts



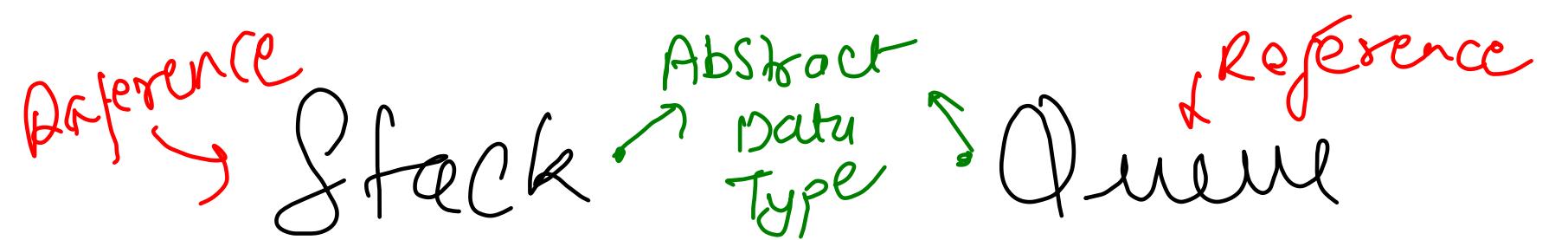
A obj = new A();
B obj = new B(); } → runtime polymorphism

A obj = new B(); } → Not possible → Why?

B obj = new A(); }

This class should be same as the reference class or it's child class.

Constructor of B never called
B's data members never initialized



→ concrete datatype
Object

→ AddLast()

→ AddLast()

→ RemoveLast()

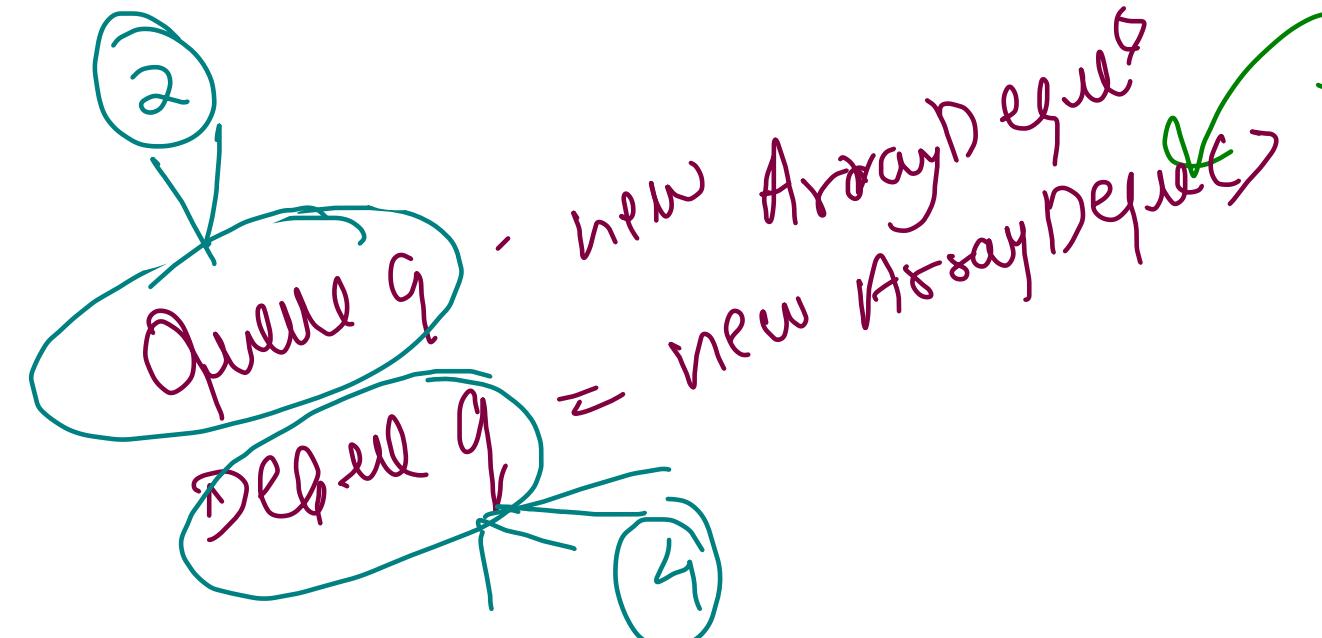
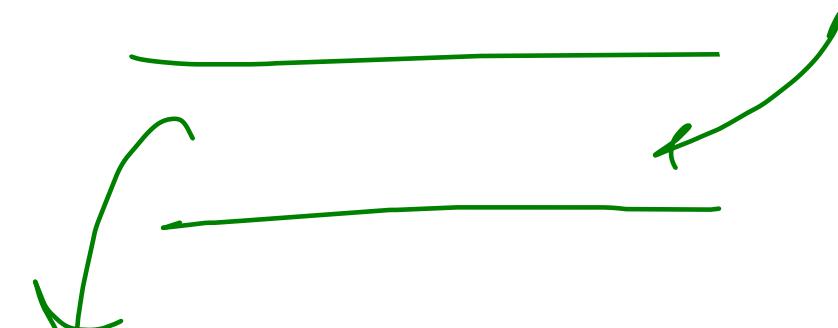
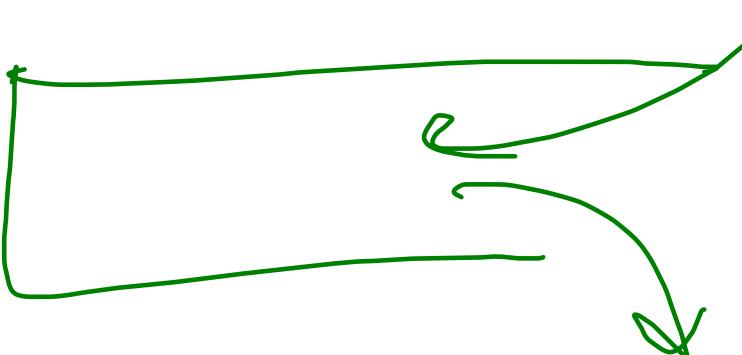
→ RemoveFirst()

→ addFirst()

→ addLast()

→ removeFirst()

→ removeLast()



Early Binding

VS

Late Binding

→ Normal Methods
of w/o inheritance}

→ Method overloading

note : → class
Static methods
Cannot be overridden
because

↳ static does not depend
on object

Whereas overriding depends
on objects type

→ Method overriding

```
class A{  
    public void earlyBind(){  
        System.out.println("Early Bind");  
    }  
  
    public void lateBind(){  
        System.out.println("Late Bind in Parent Class");  
    }  
}  
  
class B extends A{  
    @Override  
    public void lateBind(){  
        System.out.println("Late Bind in Child Class");  
    }  
}  
  
public class Main{  
    public static void main(String[] args){  
        A obj = new B();  
        obj.earlyBind(); → Early  
        obj.lateBind(); → Child  
    }  
}
```

{Abstract Data Type} ADT

Abstraction



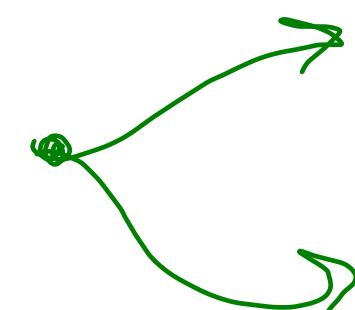
Hiding unnecessary details { complexity of class for the outside world & providing abstract view { showing necessary details }

(O-O)

→ Concrete methods are also possible

Abstract classes { Partial Abstraction }

Implementation



Interface { Full Abstraction }
(I-O-U-S)

Abstract class

→ If a class has atleast one abstract method,
then class needs to be
abstract.

only \uparrow
function prototype

not

definition.

↳ Such methods
must
be over-ridden

object/instances of abstract class
cannot be created!

Abstract class cannot have

- abstract constructor { concrete constructor can be there }
- this keyword
- abstract static methods { Note only static methods are allowed }

Abstract classes cannot be final.

Interfaces

→ class can implement multiple interfaces

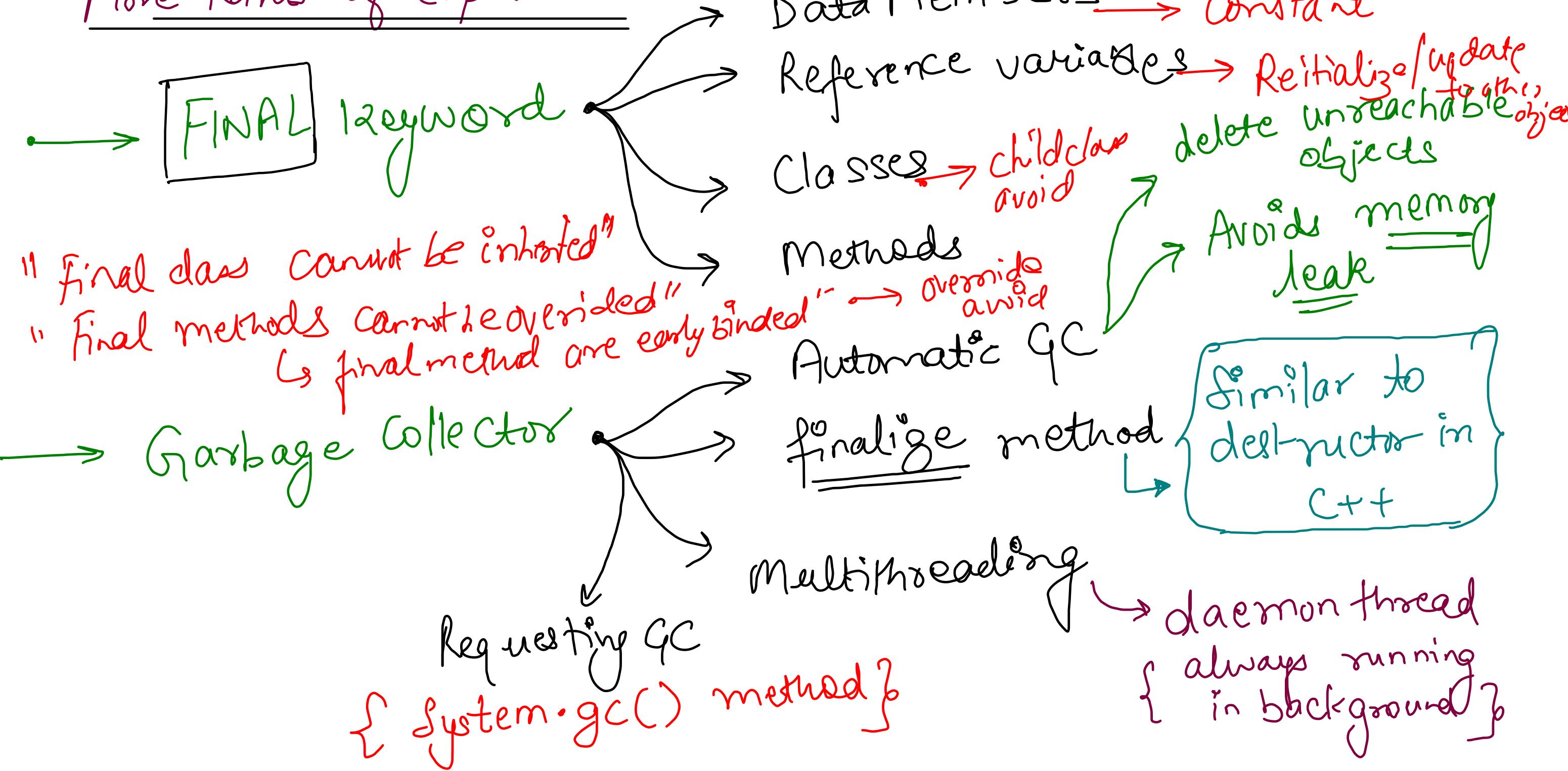
Multiple inheritance
by multiple inheritance

- "Blueprint of a class"
- all methods are abstract and public { by default }
- Interfaces cannot be instantiated !
- Variables in interface are static and final { by default }
 - default (concrete) method
 - private method
 - static method

From Java 8/9, interface can have

Interface can extend other interface # Interfaces can be nested.

More Points of Exploration



10 test case

50 nodes in each test case

Free State

↓ ↗ automatic in
Consumed state Java

if total memory \Rightarrow 500 nodes
(RAM)

6th test case

→ C++ code

Memory limit exceeded

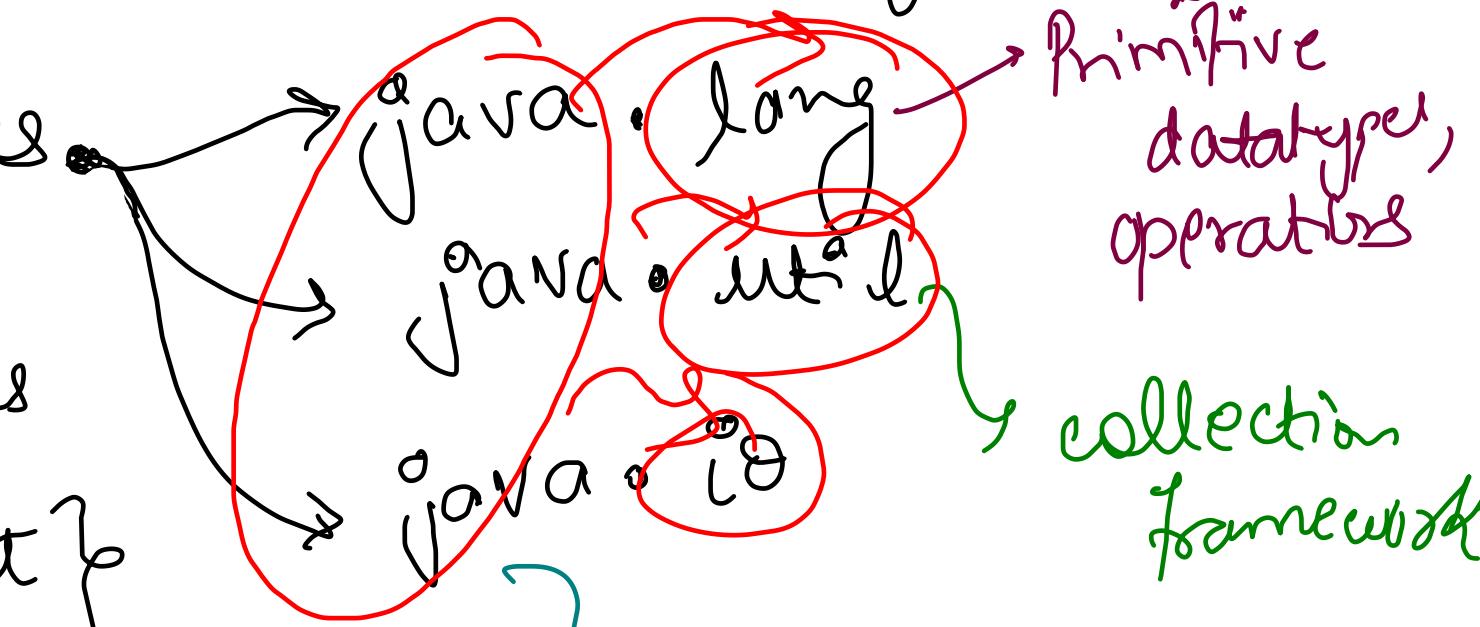
Java
no problem

Packages

grouping together classes & interfaces
or nested packages.

Built in packages

User defined packages
import statement }



primitive
datatype,
operators

collection
framework

input/output,
buffer reader, etc.

Advantages

- Avoid name collisions
- Collaborative project development
- Implement Data Hiding
- input/output from keyboard
- input/output from file

→ Singleton class Design Pattern

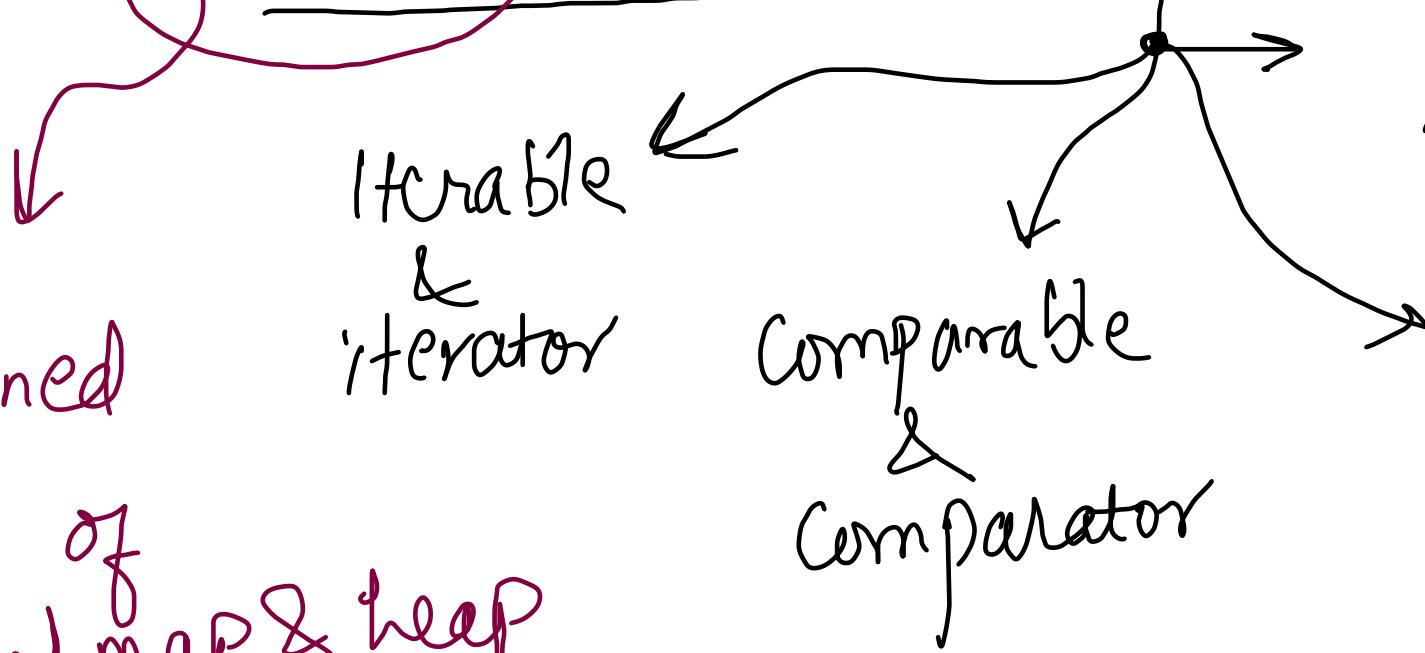
→ Private constructor

limiting the number of objects of a class
to one!

Hiding = Abstract
Encapsulation

→ Generics & collection framework

well defined
class of
hashmap & heap



Abstract Data Types (ADT)
like AL, LL, S2Q, etc.

lambda expression
(arrow function)

heap vs PQ
↓
Concrete DT → Abstract DT

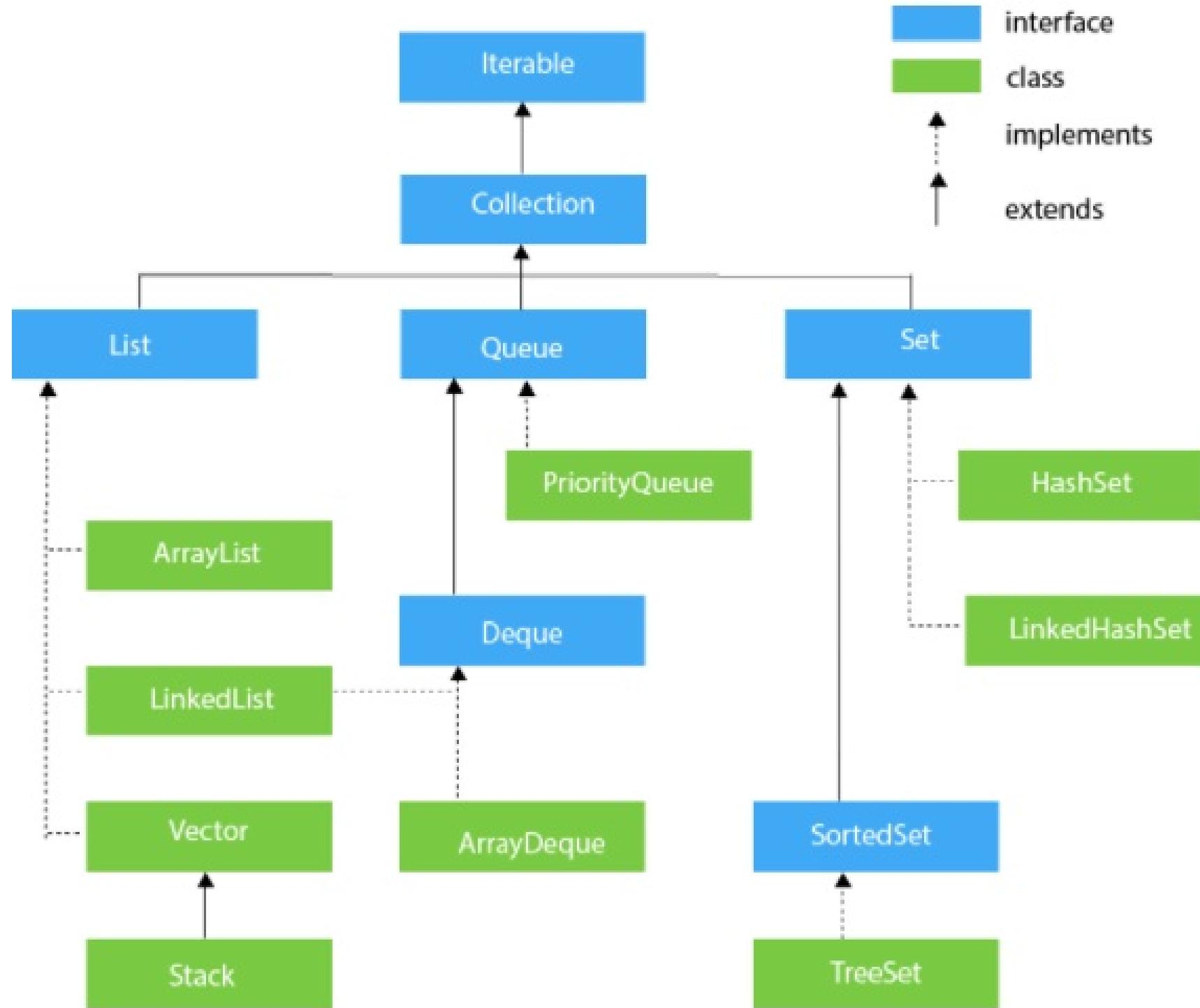
```
class Singleton{
    private Singleton(){}
    static Singleton obj;

    public static Singleton getInstance(){
        if(obj == null){
            obj = new Singleton();
        }
        return obj;
    }
}
```

```
Singleton obj1 = Singleton.getInstance();
Singleton obj2 = Singleton.getInstance();
Singleton obj3 = Singleton.getInstance();

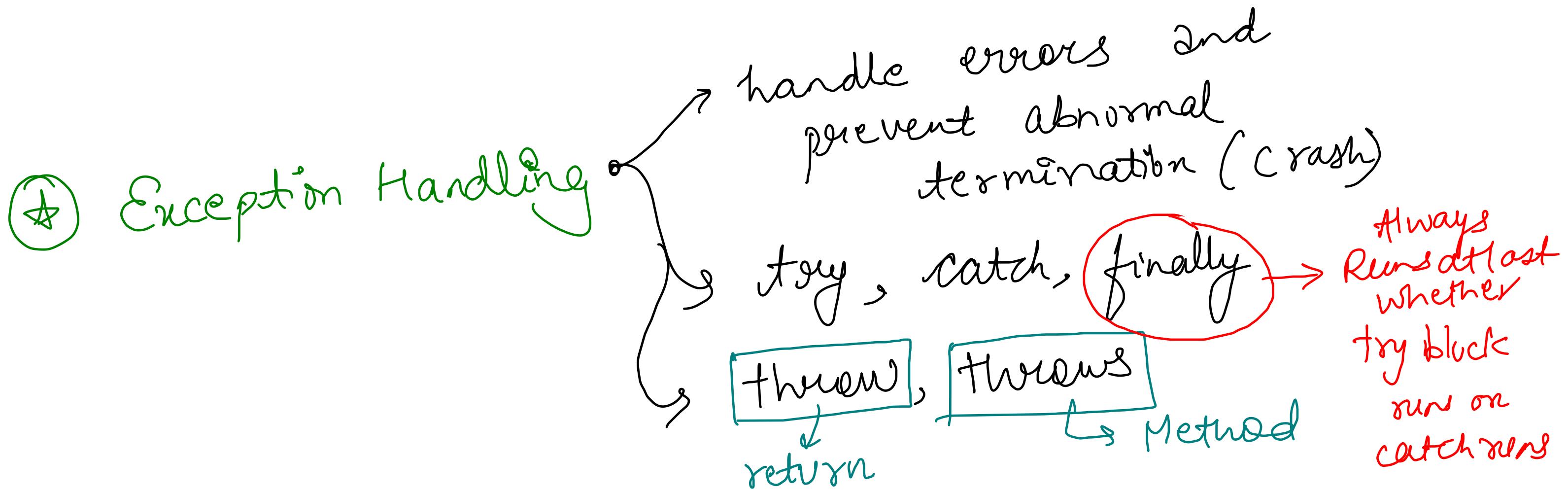
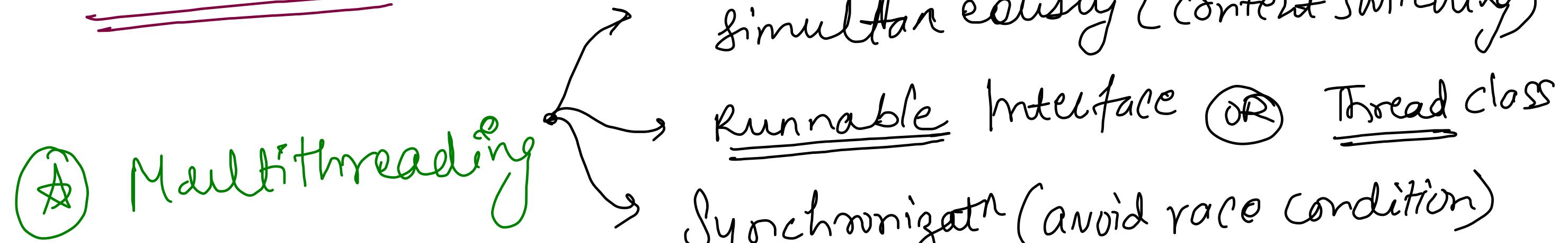
System.out.println(obj1.hashCode());
System.out.println(obj2.hashCode());
System.out.println(obj3.hashCode());
```

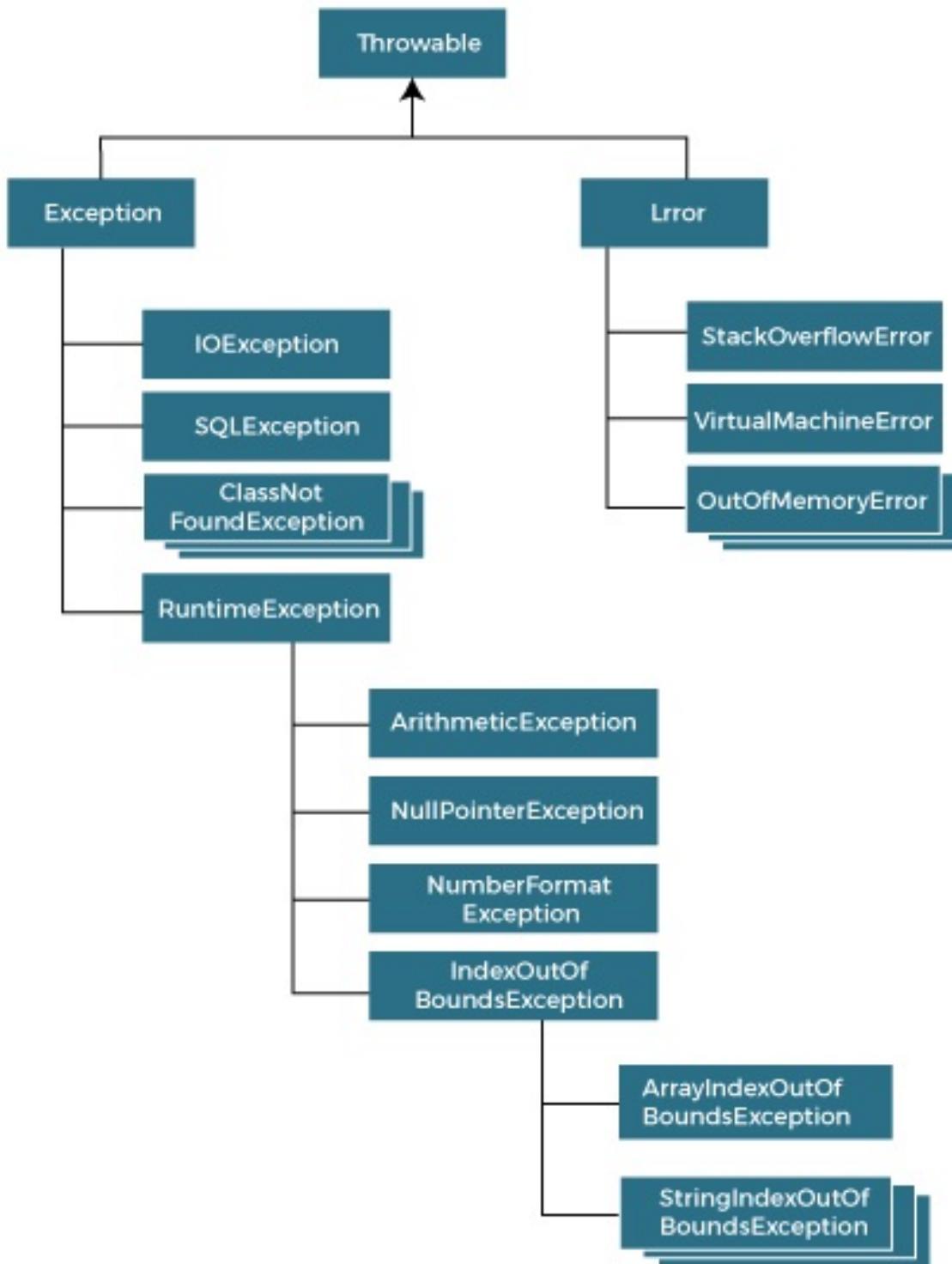
} Same hashCode



Java Collections
class
Hierarchy!

Advanced topics





Exception class Hierarchy

Types of Exceptions

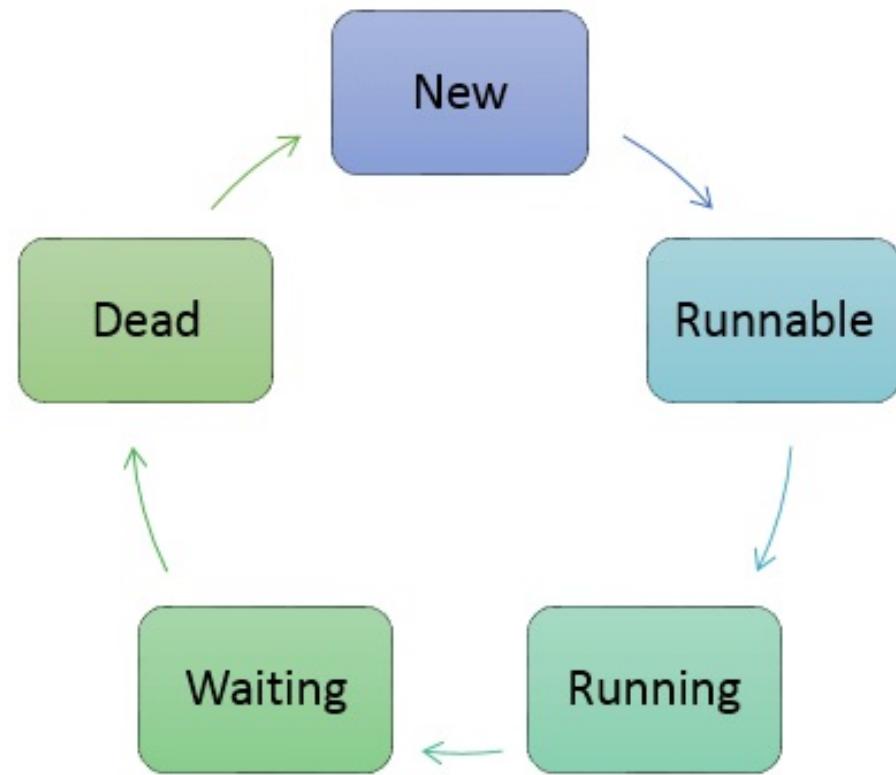
↳ Arithmetic
(Division by 0)

↳ Number Format
{ Input type mismatch }

↳ Index out of Bound

↳ Null Pointer Excep^m

↳ Stack overflow Excep^m



Thread Life Cycle in Java

1. **New:** In this phase, the thread is created using class “Thread class”. It remains in this state till the program **starts** the thread. It is also known as born thread.
2. **Runnable:** In this page, the instance of the thread is invoked with a start method. The thread control is given to scheduler to finish the execution. It depends on the scheduler, whether to run the thread.
3. **Running:** When the thread starts executing, then the state is changed to “running” state. The scheduler selects one thread from the thread pool, and it starts executing in the application.
4. **Waiting:** This is the state when a thread has to wait. As there multiple threads are running in the application, there is a need for synchronization between threads. Hence, one thread has to wait, till the other thread gets executed. Therefore, this state is referred as waiting state.
5. **Dead:** This is the state when the thread is terminated. The thread is in running state and as soon as it completed processing it is in “dead state”.

Some of the commonly used methods for threads are:

Method	Description
start()	This method starts the execution of the thread and JVM calls the run() method on the thread.
Sleep(int milliseconds)	This method makes the thread sleep hence the thread's execution will pause for milliseconds provided and after that, again the thread starts executing. This help in synchronization of the threads.
getName()	It returns the name of the thread.
setPriority(int newpriority)	It changes the priority of the thread.
yield ()	It causes current thread on halt and other threads to execute.

Implement

Stack

✓
Fixed Array ↴
Dynamic Array

Stack using

Queue

Queue

Implement

Queue

✓
Fixed Array ↴
Dynamic Array

Queue using
Stacks

→ Exception Handling.

→ Inheritance, Polymorphism

→ Access Modifiers → private
data members

(Getters & Setters)

→ Abstract
Data type

github

{ Naming
Conventions }