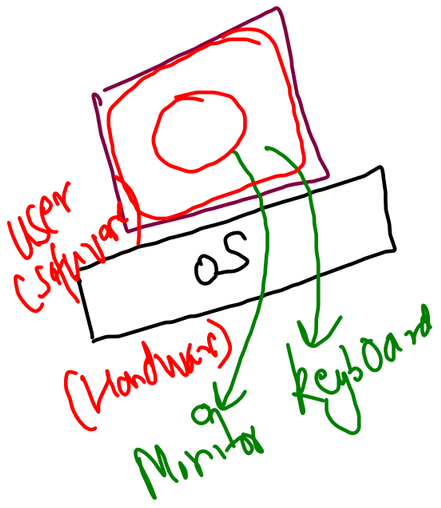
Object Oriented Programming in Java

Java Basics



→ Flow of a program

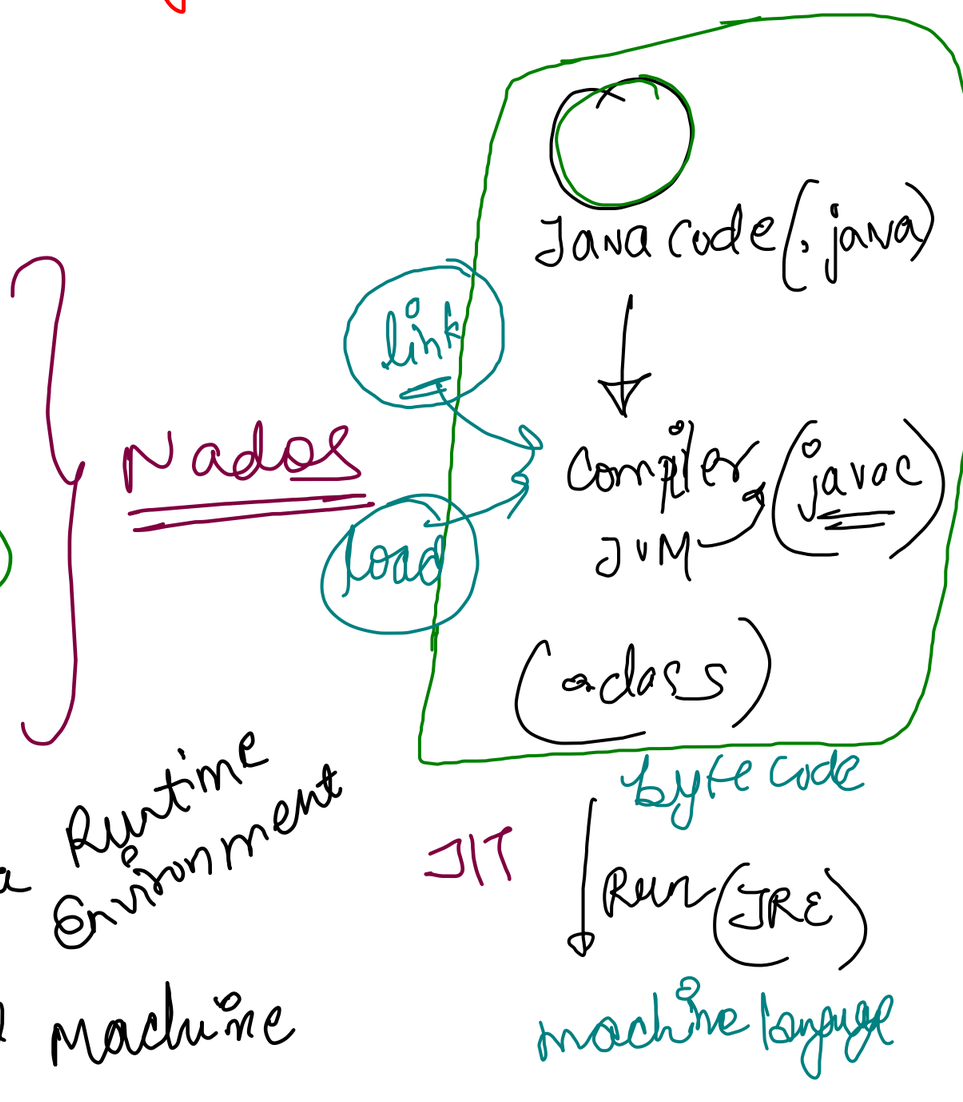
→ Why Java? (Multiplatform)

→ Architecture Java

(JDK, JVM, JRE) Java Runtime Environment

Java development kit

Java virtual machine



Pillars of Object oriented Programming

Encapsulation

- → Classes & Objects
- → Constructors {Overloading & chaining}
- → this keyword
- → Static keyword
- → Getters & Setters

Inheritance

- → IS A vs has A
- → Types of Inheritance
- → Super keyword
- → Multiple Inheritance {Diamond Problem & workaround}

Polymorphism

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

Abstraction & Data Hiding

- → Access modifiers
- → Packages
- → Abstract class
- → Interface

Object Oriented Programming | INTRODUCTION

→ Classes vs Objects

Classes → logical entity { blueprint of a object }

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

→ Data members / Instance variables → properties of objects

→ Member functions / methods → behavior of objects

} Getters & Setters

Integer.MAX_VALUE
Integer.MIN_VALUE
Integer.parseInt()

g:gp

→ 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

↳ Wrapper classes → Autoboxing, Unboxing

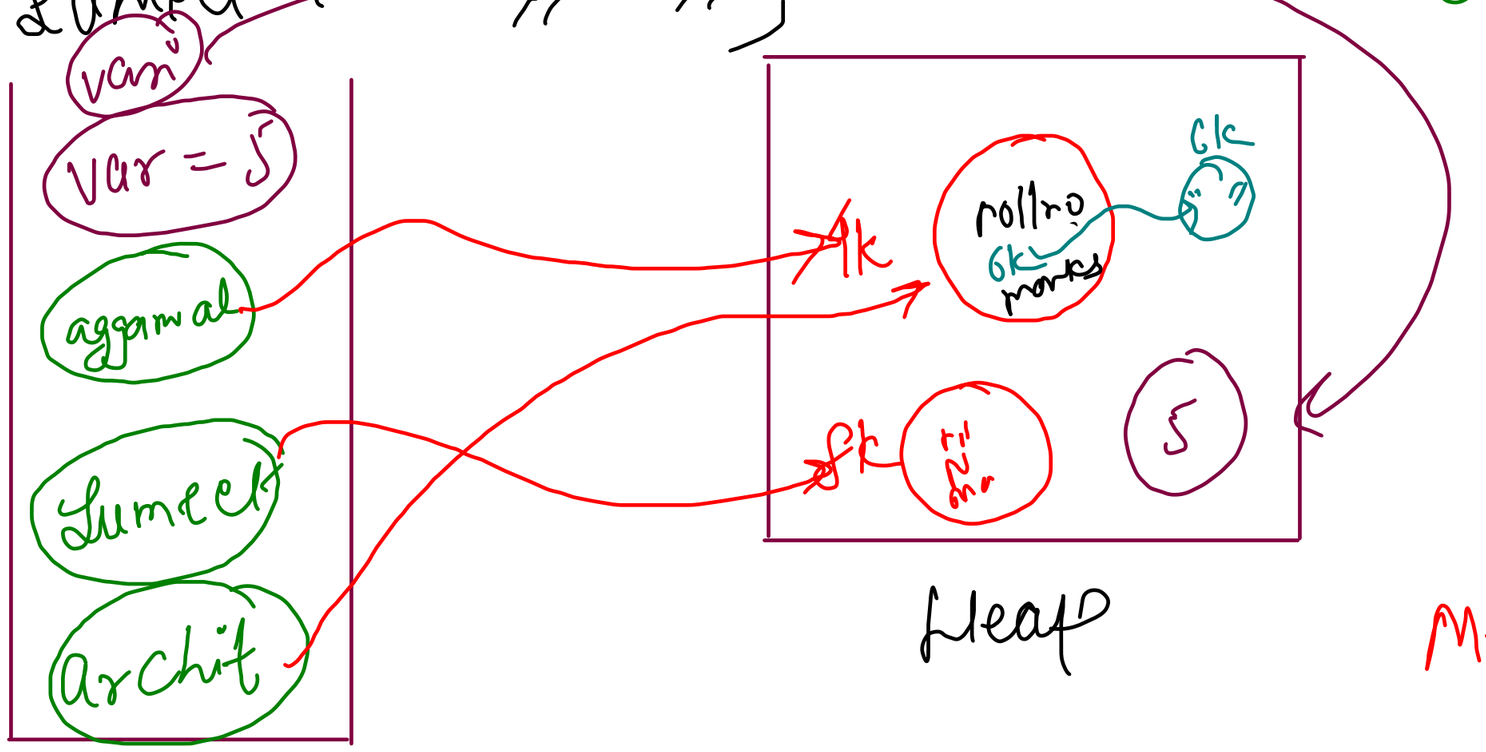
⭐ Swap Games
- Swap References
- Create temp Reference
- Create temp object

⇒ Shallow copy vs Deep copy

Student archit = new Student();

Student Lumeet = " ";

class Ref object constructor



Student aggarwal = archit;

Memory Mapping

int var = 5;

Integer vari = var; new Integer(5)

System.out.println(vari)

```java
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

```java
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

shallow copy

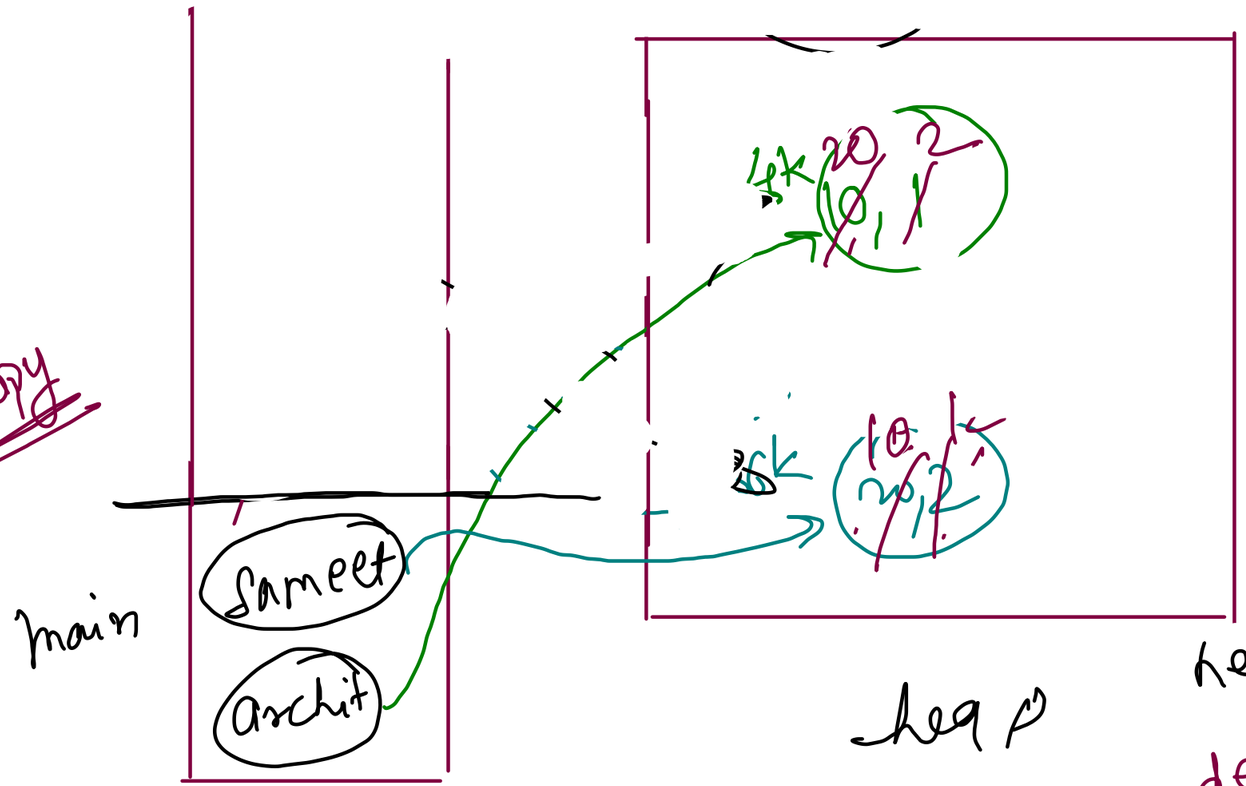
```java
public static void swap1(Student a, Student b){
    Student temp = a;
    a = b;
    b = temp;
}
public static void main(String[] args){

    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);
}
```

local member } no change



Stack

A

swap 1,2

D

swap3

deep change

deep copy

```java
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
20,2
(B)
(0,1

→ Constructor

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

} Constructor Overloading

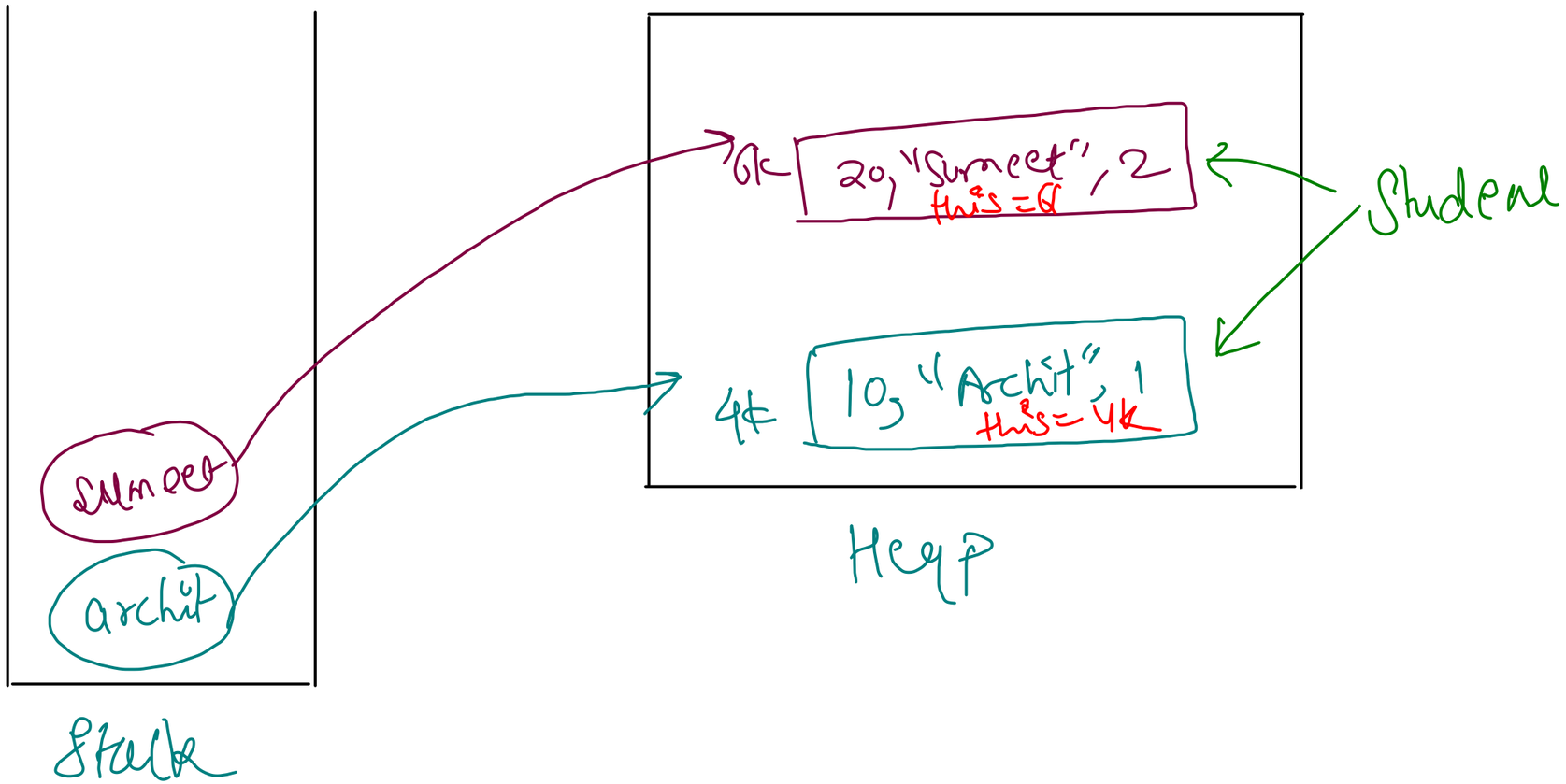
- No. of arguments
- Order of arguments
- Typecasting

→ THIS KEYWORD

⇓

Self-referential pointer

- Resolving name collision b/w data members & parameters
- Constructor chaining
- Calling current class' data members/methods
- Returning/Passing current class's object


Sumeet
archit
Stack
0x
20, "Sumeet", 2
this = 0x
4k
10, "Archit", 1
this = 4k
Student
Heap

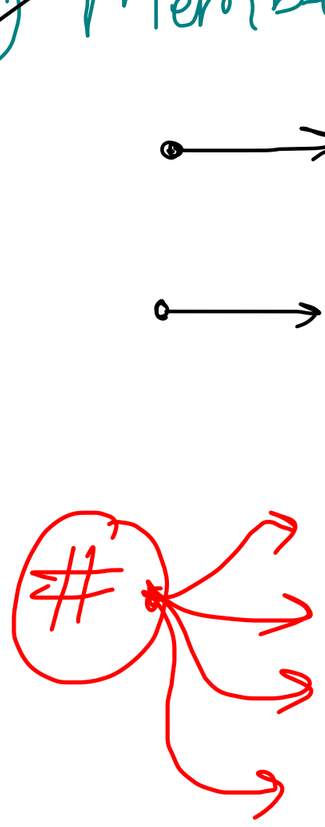
STATIC Keyword

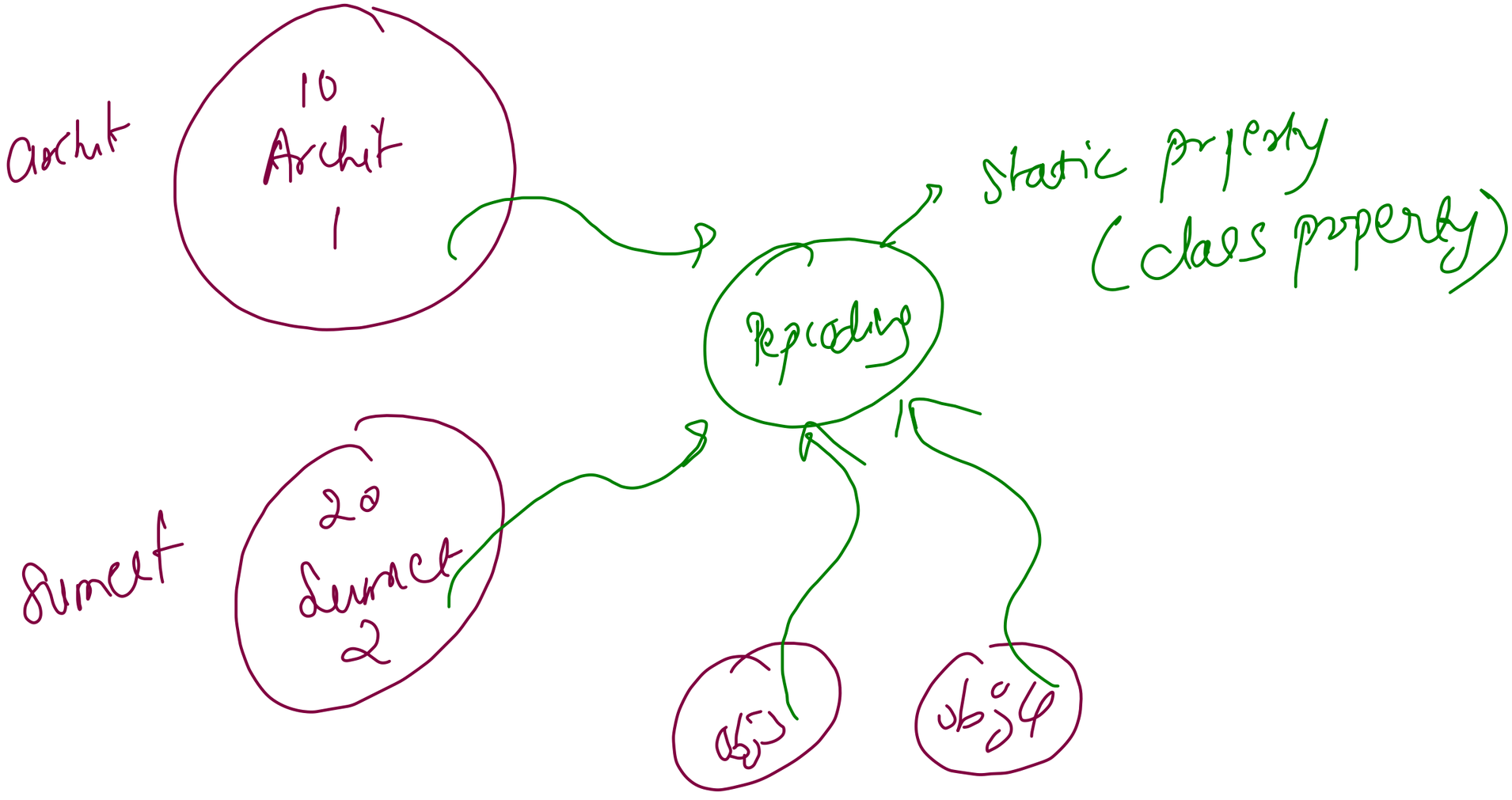
1. Data Members → Instance variables VS class variables
   → Accessing static members using class name
2. Member Functions/Methods

→ Why main() is static ?

Main class → instantiate
main() → static

→ this keyword in static method ?




Archit
10
Archit
1
static property
(class property)
Repeating
Archit
20
Archit
2
obj
obj

3 Nested class → { 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

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

Implement

Data Hiding

- Packages
- Access Modifiers
  - Public
  - Private
  - Protected
  - Default

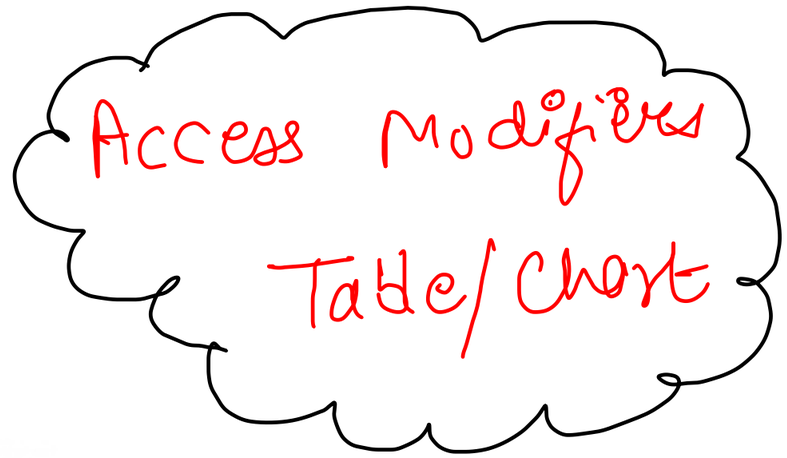
Getters & Setters

class → object

private property

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

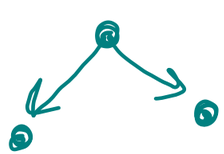
↳ Parent child → child class
superclass → subclass

Types of Inheritance

- Single Inheritance

- Multi-level Inheritance

- Heirarchial Inheritance

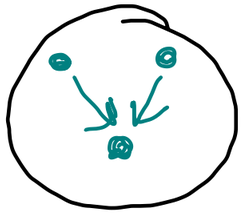


SUPER keyword

refers immediate parent class' object

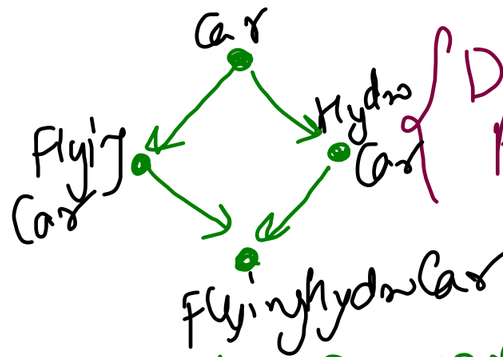
- Constructor chaining in Inheritance
- access data members/ methods of parent class.

Multiple Inheritance



→ Why not possible in Java?

- Hybrid Inheritance



{Diamond Problem}

Parent class methods collision resolution?

Flying, Hyde
↳ gearchange()

Solution

↳ multiple interfaces can be implemented by a single class.

Object class {Superclass of all classes}

→ What type of Inheritance? ~~Multi-level~~

→ methods in object class

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

is A relationship
{Inheritance}

VS

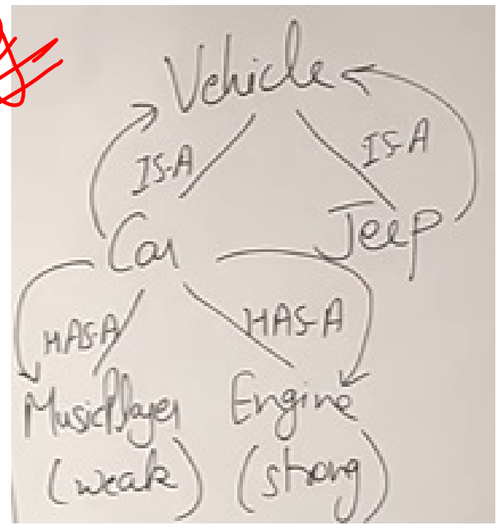
has A relationship
{Association}

→ Tightly coupled Relationship → Direct access through super keyword

→ loosely coupled Relationship → Accessing using Object

Example → Nested class
→ Object Member

Eg



Types of Association (has A)

Aggregation (Weak)

Composition (Strong)

Polymorphism → one action in many ways/forms

Types of Polymorphism

Compile-Time Polymorphism
or
Static polymorphism

(single class 2 functions)

{ 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

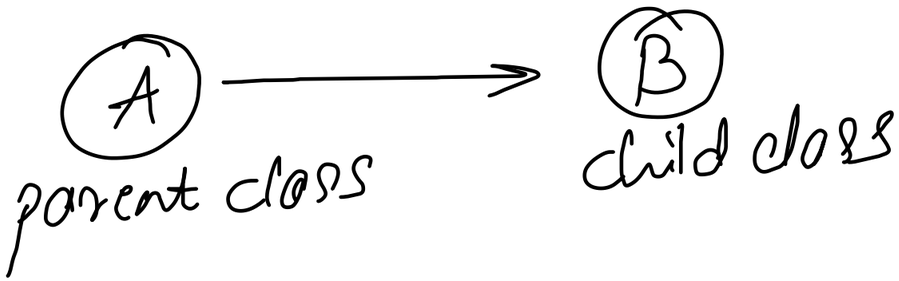
- Same
  - No of arguments
  - Type of arguments
  - Return type

ClassName referenceName = new ClassName();

→ gives the functions which we can call

ClassName();

→ provides the implementation of those functs



parent class

child class

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

A obj = new A();

B obj = new B();

A obj = new B(); → Runtime polymorphism

✓ Possible

B obj = new A();

⇒ Not possible → Why?

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

Reference → Stack ← Abstract Data Type → Queue ← Reference

Deque ← Object ← concrete datatype

- addLast()
- addLast()
- removeFirst()
- removeLast()

- addFirst()
- addLast()
- removeFirst()
- removeLast()

2

Queue q = new ArrayDeque()

Deque q = new ArrayDeque()

4

Early Binding vs Late Binding

→ Normal Methods
(w/o inheritance)

→ Method overloading
→ class

→ Method Overriding

note :→ Static methods cannot be overrided → object

because
↳ Static does not depend on object
whereas overriding depends on objects type

```java
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 Datatype} (ADT)

Abstraction

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

Implementation

- Abstract classes {Partial Abstraction} (0-100)
  - ↳ Concrete methods are also possible
- Interface {Full Abstraction} (100%)

Abstract class

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

↓ only function prototype not definition.

↳ Such methods must be over-riden

objects/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)

→ "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}

From Java 8/9, interface can have
- default (concrete) method
- private method
- static method

Inteface can extend other intyface # Interfaces can be nested.

More Points of Exploration

→ FINAL keyword

- Data Members → constant
- Reference variables → Reitialize/update to another object
- Classes → childclass avoid
  - "Final class cannot be inherited"
- Methods → override avoid
  - "Final methods cannot be overrided"
    - ↳ final method are early binded

→ Garbage Collector

- Automatic GC
  - delete unreachable objects
  - Avoids memory leak
- finalize method
  - { Similar to destructor in C++ }
- Requesting GC
  - { System.gc() method }
- Multithreading
  - daemon thread
    - { always running in background }

10 test case

100 node in each test case

free state

automatic in java

consumed state

if total memory (RAM) ⇒ 500 nodes

6th test case

C++ code

Memory limit Exceeded

Java

no problem

Packages

- grouping together classes & Interfaces or nested packages.
- Built in packages
  - java.lang → Primitive datatypes, operations
  - java.util → collection framework
  - java.io → input/output, BufferReader, etc.
- User defined packages {import statement}

Advantages

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

→ Singleton class Design Pattern

- Private Constructor
- limiting the number of objects of a class to one!

Hiding ≈ Abstraction + Encapsulation

heap vs PQ

- heap → Concrete DT
- PQ → Abstract DT

→ Generics & collection framework

- Abstract Data Types (ADT) like AL, LL, S&Q, etc.
- Iterable & iterator
- Comparable & Comparator
- lambda expression (arrow function)
- user defined class of hashmap & heap

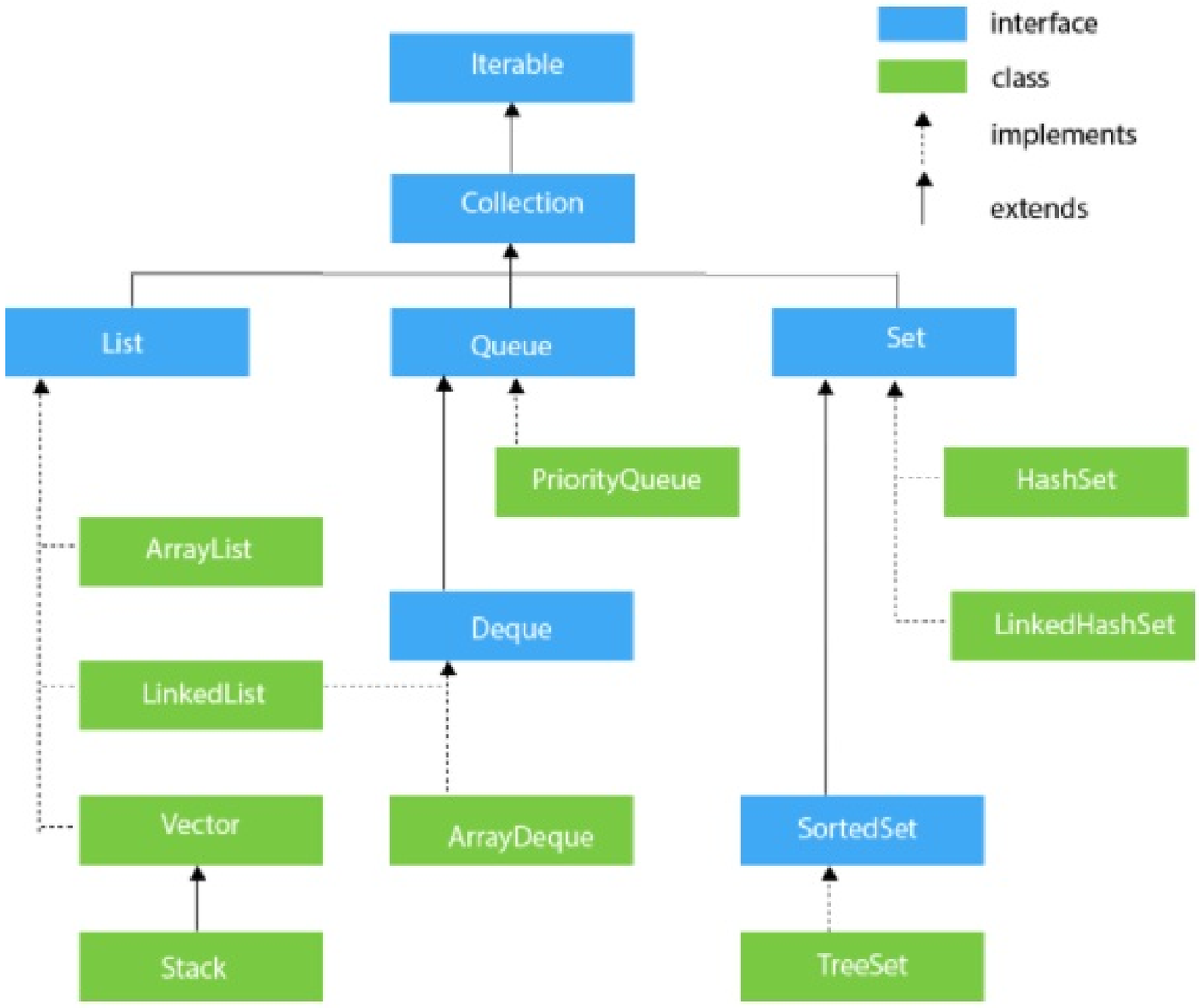
```java
class Singleton{
    private Singleton(){}
    static Singleton obj;

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

```java
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!

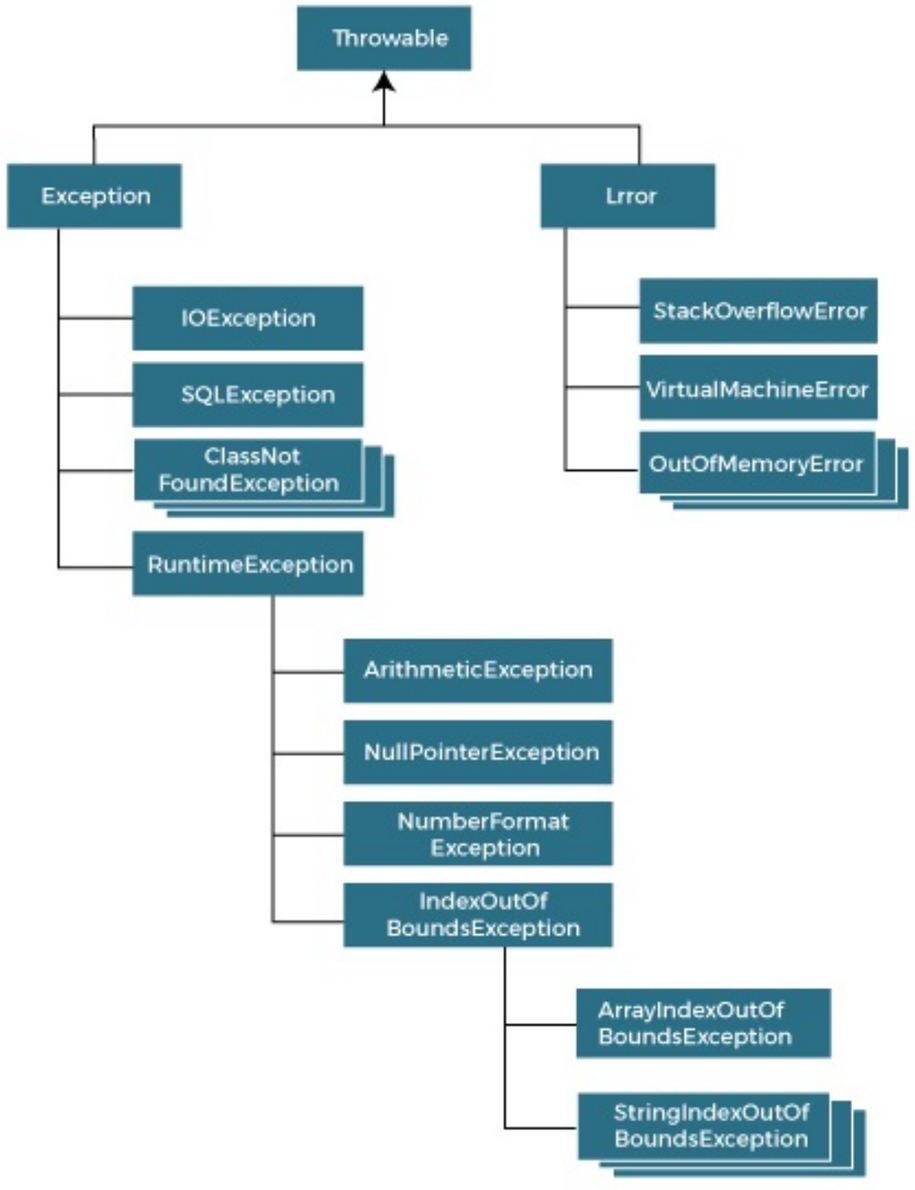
Advanced topics

⭐ Multithreading

- Executing multiple threads simultaneously (context switching)
- Runnable Interface (OR) Thread class
- Synchronizatn (avoid race condition)

⭐ Exception Handling

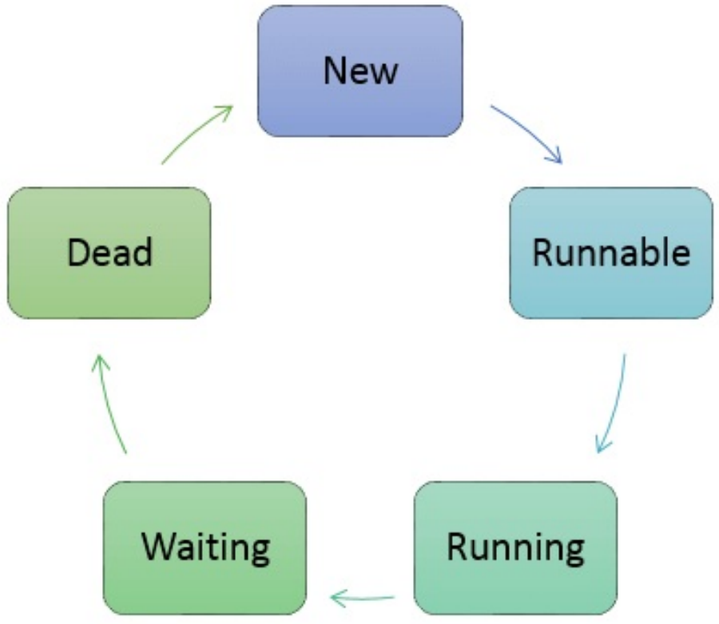
- handle errors and prevent abnormal termination (crash)
- try, catch, finally → Always Runs at last whether try block run or catch runs
- throw, throws
  - throw → return
  - throws → Method



Exception class Heirarchy

Types of Exceptions

- ↳ Arithmetic (Division by 0)
- ↳ Number Format {Input type mismatch}
- ↳ Index out of Bound
- ↳ Null Pointer Excepn
- ↳ Stack overflow Excepn



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 ✓

Implement Queue

- Fixed Array ✓
- Dynamic Array ✓

Implement Deque

Stack using Queue

Queue using Stacks

github
- Naming Convention

→ Exception Handling

→ Inheritance, Polymorphism

→ Access Modifiers → private data members (Getters & Setters)

→ Abstract Data type