Object Oriented Programming

OOP 1 → Classes, objects, constructors and keywords.

① Classes and Objects

A class is a template for an object, and an object is an instance of a class.
A class creates a new data type that can be used to create objects.

When you declare an object of a class, you are creating an instance of that class.
Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)

② properties of objects

Objects are characterized by three essential properties: state, identity, and behavior.
The state of an object is a value from its data type. The identity of an object distinguishes one object from another.
It is useful to think of an object's identity as the place where its value is stored in memory.
The behavior of an object is the effect of data-type operations.

③ Objects and Instance Variables and how to access them →

Variables inside the class.

The dot operator links the name of the object with the name of an instance variable.
Although commonly referred to as the dot operator, the formal specification for Java categorizes the . as a separator.

Inside the Class.

Instance Variables

Student

Roll number, Name, Marks.

14 Kunal 87.9

28 Rahul 92.7

33 Anais 95.38

Student 1

Student 2

Student 3

Object / reference Variable

```
        Student[] students = new Student[5];

        // just declaring
        Student kunal;

        System.out.println(Arrays.toString(students));
    }
}

// create a class
// for every single student
class Student {
    int rno;
    String name;
    float marks;
}
```

Printing Kunal will give output as 'null'.

→ Even printing array of students will print 5 null(s).

(H)

The 'new' keyword dynamically allocates(that is, allocates at run time)memory for an object & returns a reference to it.
This reference is, more or less, the address in memory of the object allocated by new.
This reference is then stored in the variable.
Thus, in Java, all class objects must be dynamically allocated.

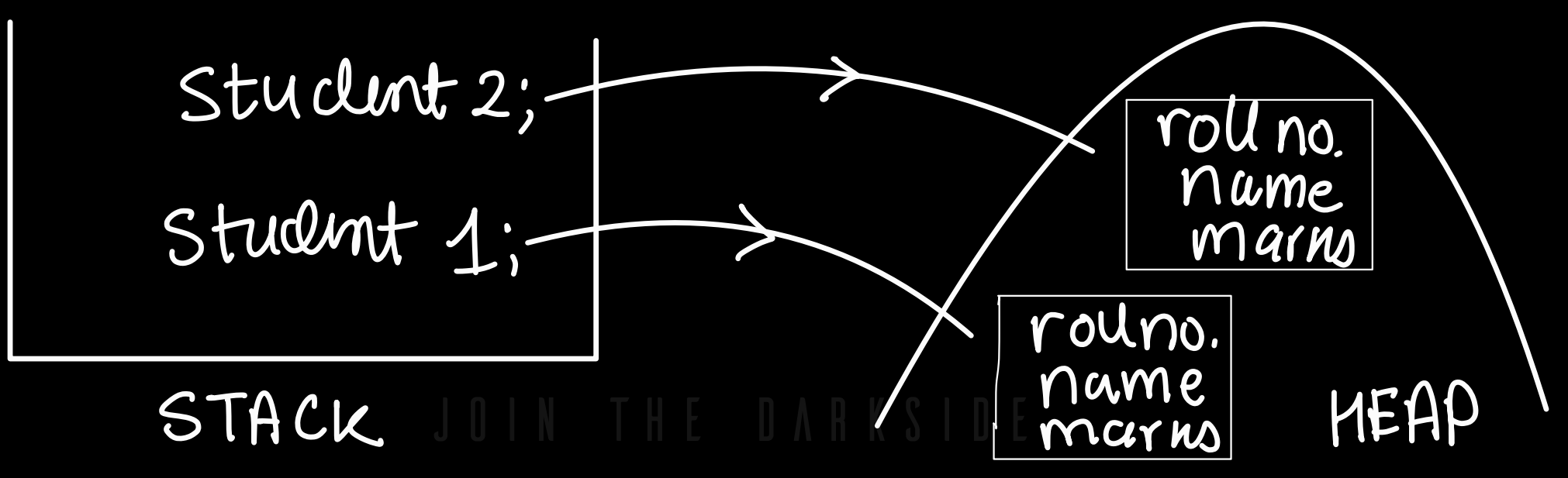
The new keyword →

```
Student student1;          // declaring Variable
student1 = new student();
```

↳ dynamically allocates memory for an object and returns reference to it.

```
Student student1 = new student();
```

compile time | run time



This is how dynamic Memory allocation works

It is important to understand that new allocates memory for an object during run time.

```
Box b1 = new Box();
Box b2 = b1;
```

b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part
of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object
through b2 will affect the object to which b1 is referring, since they are the same object.
When you assign one object reference variable to another object reference variable, you are not creating a copy of the
object, you are only making a copy of the reference.

Declaring the object and then allocating object; Why we cannot change reference in Java!

```
Box mybox; // declare reference to object
     mybox = new Box(); // allocate a Box object
```

The first line declares mybox as a reference to an object of type Box. At this point, mybox does not yet refer to an
actual object. The next line allocates an object and assigns a reference to it to mybox. After the second line executes,
you can use mybox as if it were a Box object. But in reality, mybox simply holds, in essence, the memory address of the
actual Box object.
The key to Java's safety is that you cannot manipulate references as you can actual pointers.
Thus, you cannot cause an object reference to point to an arbitrary memory location or manipulate it like an integer.

This is how to declare →

```
        Student kunal = new Student();

        kunal.rno = 13;
        kunal.name = "Kunal Kushwaha";
        kunal.marks = 88.5f;

        System.out.println(kunal.rno);
        System.out.println(kunal.name);
        System.out.println(kunal.marks);
    }
}
```

Output →

13
Kunal Kushwaha
88.5f

default value is 0, null, 0.0

Imp ★ →

If there is already a default value set in the class itself as shown below as a code output example

```
// create a class
// for every single student
class Student {
    int rno;
    String name;
    float marks = 90;
}
```

Output

13
Kunal Kushwaha
90.0

It will not print 88.5 as we have commented that line.

kunal.marks = 88.5f; X

It will print the default value

⑤ Constructors

A Closer Look at new:
classname class-var = new classname ( );
Here, class-var is a variable of the class type being created. The classname is the name of the class that is being
instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines
what occurs when an object of a class is created.

Constructor is a special function that runs when you create an object and also allocates some variables

student student1 = new student();

↓

This is a by default constructor even when no constructor is present by default.

Class w/o Constructor

```
Class student {
    int rollno.;
    String name;
}
```

Class with constructor

```
Class student {
    int roll no.;
    String name;
}
  student () {
    this.r no = 13;
    this. name = "Lak";
}
```

```java
// create a class
// for every single student
class Student {
    int rno;
    String name;
    float marks = 90;

    // we need a way to add the values of the above
    // properties object by object

    // we need one word to access every object

    Student () {
        this.rno = 13;
        this.name = "Kunal Kushwaha";
        this.marks = 88.5f;
    }
}
```

This is what a constructor looks like

↓

'This' keyword is replaced by Student 1, Student 2.

Example →

Student kunal =

new student (13, "kunal k.", 88.5f);

This keyword in the constructor calls all the function values (r.no, name) along with the object variable. (kunal in this case)

```java
Student kunal = new Student();

//      kunal.rno = 13;
//      kunal.name = "Kunal Kushwaha";
//      kunal.marks = 88.5f;

System.out.println(kunal.rno);
System.out.println(kunal.name);
System.out.println(kunal.marks);
}
```

→ It will become like this.

Lakshay is an object of class Student

assigning all the required values to the constructor

```
Student lakshay = new Student( roll: 13, naam: "name", perc: 99.54f);
System.out.println(lakshay.rno);
System.out.println(lakshay.name);
System.out.println(lakshay.marks);
}

}

class Student {
    int rno;
    String name;
    float marks;

    Student(int roll,String naam,float perc) {
        this.rno = roll ;
        this.name = naam;
        this.marks = perc;
    }
}
```

→ printing all

Constructor of type Student taking values like a function

name of these parameters should be same as present in the class above

These parameters takes values from Object declaration and get assigned to classes

⇒ If we add a function in our class ↓

```
Void greeting () {
    System.out.println ("Hello" + name);
    }
```

↳ Output will be - hello kunal

But we may use this.name operator.

```
Void greeting () {
    System.out.println ("Hello" + this.name);
    }
```

↳ using this will help us in differentiating between diff objects

eg- Student 1, Student 2.
(will refer to the current object).

The this Keyword:
Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword.
this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked.

Internally, for every object, this keyword specifies the roll no., name for every object

whenever we do

Lakshay.rno = 13; → this.roll no. is called and this is replaced by Lakshay

Constructor Overloading

if Student kunal = new Student();

now, as (); is empty, it will go to the empty constructor, (which doesn't takes these values along with it like String naam, int age... etc.)

if student kunal = new Student("13, "kunal", 98);

It will go to the constructor which takes 3 values and has 3 parameters

→ For the compilation to be successful each constructor must contain a different list of arguments.

Calling a Constructor from another Constructor

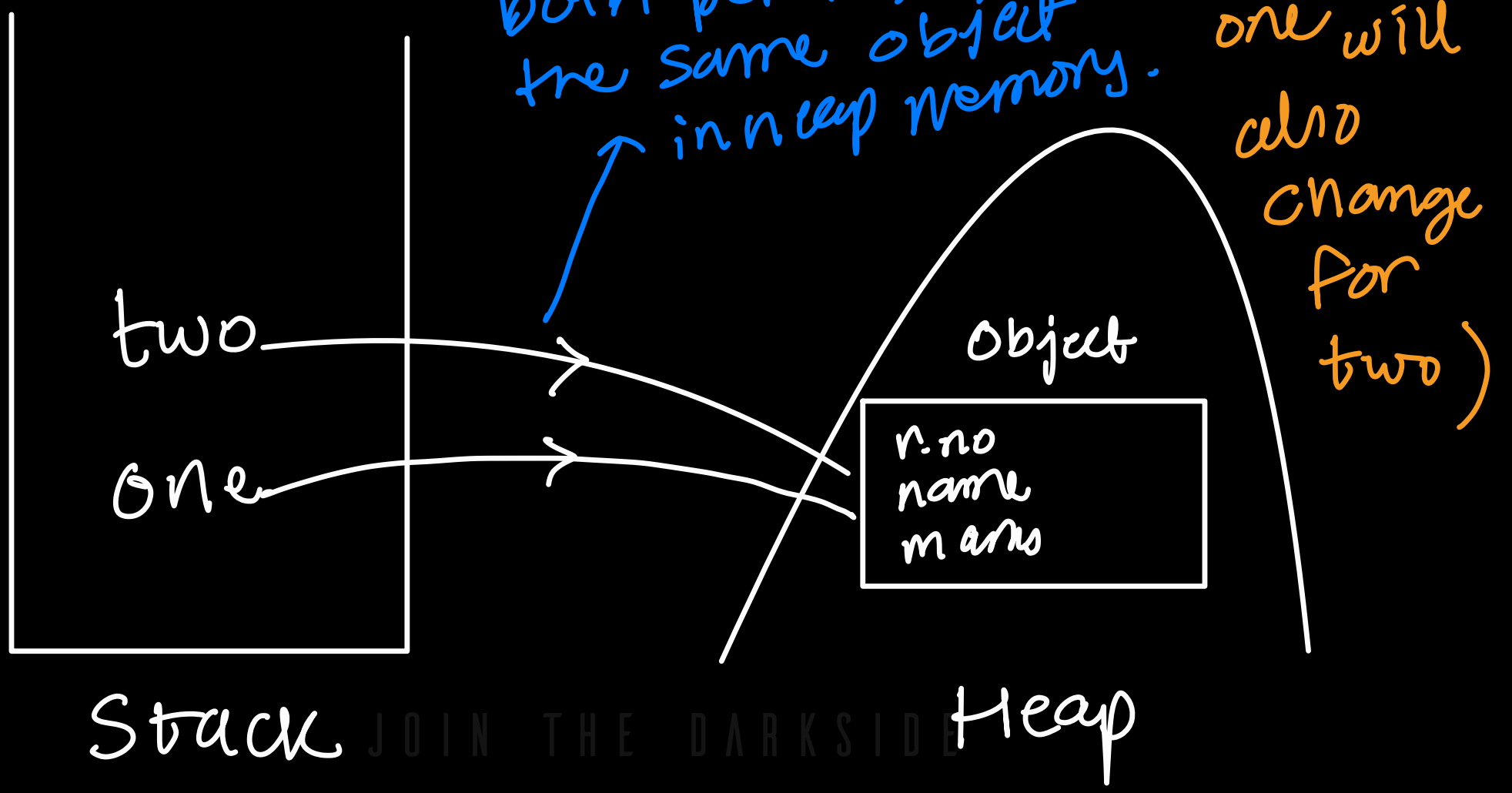
```
Student Random = new Student();
```

```
    }

    Student () {
        // this is how you call a constructor from another constructor
        // internally: new Student (13, "default person", 100.0f);
        this (rno: 13, name: "default person", marks: 100.0f);
    }

    // Student arpit = new Student(17, "Arpit", 89.7f);
    // here, this will be replaced with arpit
    Student (int rno, String name, float marks) {
        this.rno = rno;
        this.name = name;
        this.marks = marks;
```

calls this constructor.

```
Student one = new Student();
Student two = one;
```

(change in one will also change for two)



Constructors:

Once defined, the constructor is automatically called when the object is created, before the new operator completes.
Constructors look a little strange because they have no return type, not even void.
This is because the implicit return type of a class' constructor is the class type itself.

In the line
Box mybox1 = new Box();
new Box( ) is calling the Box( ) constructor.

Inheritance and constructors in Java:

In Java, constructor of base class with no argument gets automatically called in derived class constructor.
For example, output of following program given below is:

Base Class Constructor Called
Derived Class Constructor Called

```
 // filename: Main.java
class Base {
  Base() {
    System.out.println("Base Class Constructor Called ");
  }
}

class Derived extends Base {
  Derived() {
    System.out.println("Derived Class Constructor Called ");
  }
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived();
  }
}
```

Any class will have a default constructor, does not matter if we declare it in the class or not. If we inherit a class,
then the derived class must call its super class constructor. It is done by default in derived class.
If it does not have a default constructor in the derived class, the JVM will invoke its default constructor and call
the super class constructor by default. If we have a parameterised constructor in the derived class still it calls the
default super class constructor by default. In this case, if the super class does not have a default constructor,
instead it has a parameterised constructor, then the derived class constructor should call explicitly call the
parameterised super class constructor.

⑥ Wrapper Classes

`int a = 10;` ⟶ primitive data type

`Integer num = 45;`

↳ Wrapper Class

∴ now we can use num.xxx = yyy; to convert Int to object

↳ functions like compareTo; longValue;

⑦ Final Keyword

`final int INCREASE = 2;`

↓

convention is to use capital always.

Value can't be MODIFIED

As we cannot modify it ∴ always Initialise the final keyword.

When final keyword is used before primitive, it cannot be modified, but when used before reference var.

eg-

final Student kunal = new Student();

kunal.name = "new name";

this is allowed

✗ kunal = other object;

this is not possible, can't be reassigned.

```java
class A {
    final int num = 10;
    String name;

    public A(String name) {
        this.name = name;
    }
}
```

```java
    final A kunal = new A(name: "Kunal Kushwaha");
    kunal.name = "other name";

    // when a non primitive is final, you cannot reassign it.
    kunal = new A(name: "new object");
}

static void swap(Integer a, Integer b) {
    Integer temp = a;
    a = b;
    b = temp;
}
```

→ allowed

→ not allowed will give error.

final Keyword:
A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant.
This means that you must initialize a final field when it is declared.

It is a common coding convention to choose all uppercase identifiers for final fields:

```
final int FILE_OPEN = 2;
```

Unfortunately, final guarantees immutability only when instance variables are primitive types, not reference types.
If an instance variable of a reference type has the final modifier, the value of that instance variable (the reference
to an object) will never change—it will always refer to the same object—but the value of the object itself can change.

⑧ Garbage Collection

Java does that automatically
we can use finalize method
and can be called by java when
it does garbage collection

The finalize( ) Method:
Sometimes an object will need to perform some action when it is destroyed.
To handle such situations, Java provides a mechanism called finalization. By using finalization,
you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
To add a finalizer to a class, you simply define the finalize( ) method. The Java run time calls that method whenever
it is about to recycle an object of that class. Right before an asset is freed, the Java run time calls the finalize( )
method on the object.

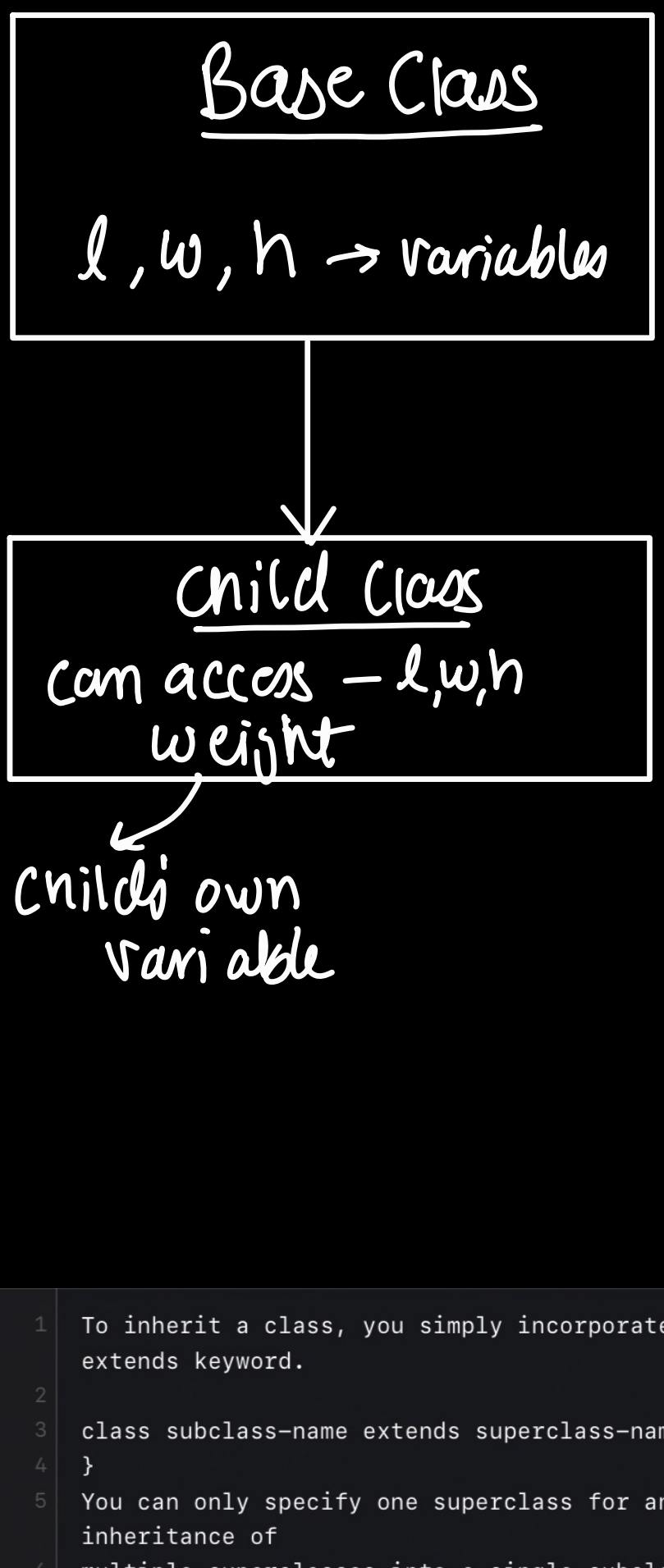
```
protected void finalize( ) {
    // finalization code here
}
```

Object oriented Programming III

Principles - Inheritance, Polymorphism, Encapsulation, Abstraction.

(I) Inheritance

Child class will have properties of its own + it can access the properties (variables) of the base class.



Child class is inheriting properties from the base class.

```
Class Child extends Base {
    int weight;
}
Child baby = new Child();
baby.l;
baby.w;
baby.h;
```

Can access all these elements

```
To inherit a class, you simply incorporate the definition of one class into another by using the
extends keyword.

class subclass-name extends superclass-name { // body of class
}
You can only specify one superclass for any subclass that you create. Java does not support the
inheritance of
multiple superclasses into a single subclass. You can, as stated, create a hierarchy of inheritance in
which a subclass
becomes a superclass of another subclass. However, no class can be a superclass of itself.

Although a subclass includes all of the members of its superclass, it cannot access those members of
the superclass
that have been declared as private.

```

how it works →

BoxWeight.java

```java
package com.kunal.properties.inheritance;

public class BoxWeight extends Box{
    double weight;

    public BoxWeight() {
        this.weight = -1;
    }

    public BoxWeight(double l, double h, double w, double weight) {
        super(l, h, w); // what is this? call the parent class constructor
        // used to initialise values present in parent class
        this.weight = weight;
    }
}
```

extends keyword — inherits the Box class

a simple constructor

always important to initialise this super(l,h,w) first

This constructor takes 4 values, 3 using super keyword

Main.java

```java
package com.kunal.properties.inheritance;

public class Main {
    public static void main(String[] args) {
//        Box box1 = new Box(4.6, 7.9, 9.9);
//        Box box2 = new Box(box1);
//        System.out.println(box1.l + " " + box1.w + " " + box1.h);

        BoxWeight box3 = new BoxWeight();
        BoxWeight box4 = new BoxWeight( l: 2,  h: 3,  w: 4,  weight: 8);
        System.out.println(box3.h + " " + box3.weight);
    }
}
```

Main.java → calls and prints

this constructor super(l,h,w) calls this

Box.java

```java
package com.kunal.properties.inheritance;

public class Box {
    private double l;
    double h;
    double w;
//    double weight;

    static void greeting() {
        System.out.println("Hey, I am in Box class. Greetings!");
    }

    public double getL() {
        return l;
    }

    Box () {
        this.h = -1;
        this.l = -1;
        this.w = -1;
    }

    // cube
    Box (double side) {
        // super(); Object class
        this.w = side;
        this.l = side;
        this.h = side;
    }

    Box(double l, double h, double w) {
        System.out.println("Box class constructor");
        this.l = l;
        this.h = h;
        this.w = w;
    }
```

Box class, base class for the child class boxweight has various constructors

if a variable is declared as private, it can't be used by the child class.

```
  Box box5 = new BoxWeight(2, 3, 4, 8);
  System.out.println(box5.w);

// there are many variables in both parent and child classes
// you are given access to variables that are in the ref type i.e. BoxWeight
// hence, you should have access to weight variable
// this also means, that the ones you are trying to access should be initialised
// but here, when the obj itself is of type parent class, how will you call the constructor of child class
// this is why error
  BoxWeight box6 = new Box(2, 3, 4);
  System.out.println(box6);
```

This can access l,w,h but not weight, even when wt is being initialised → This is because this reference variable defines not the new BoxWt();

Can't access (l,w,h) because they were never initialised. as Boxweight is the defining one

```
 A Superclass Variable Can Reference a Subclass Object:
 It is important to understand that it is the type of the reference variable—not the type of the object
 that it refers
 to—that determines what members can be accessed.
 When a reference to a subclass object is assigned to a superclass reference variable, you will have
 access only to
 those parts of the object defined by the superclass.

 plainbox      =  weightbox;
 (superclass)     (subclass)

 SUPERCLASS ref = new SUBCLASS();    // HERE ref can only access methods which are available in
 SUPERCLASS
```

Uses of Super keyword →

1)

```
 Using super:
 Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword
 super.
 super has two general forms. The first calls the superclass' constructor.
 The second is used to access a member of the superclass that has been hidden by a member of a
 subclass.

 BoxWeight(double w, double h, double d, double m) {
     super(w, h, d); // call superclass constructor
     weight = m;
 }

 Here, BoxWeight( ) calls super( ) with the arguments w, h, and d. This causes the Box constructor to
 be called,
 which initializes width, height, and depth using these values. BoxWeight no longer initializes these
 values itself.
 It only needs to initialize the value unique to it: weight. This leaves Box free to make these values
 private if desired.

 Thus, super( ) always refers to the superclass immediately above the calling class.
 This is true even in a multileveled hierarchy.

 class Box {
      private double width;
      private double height;
      private double depth;

      // construct clone of an object
```

```
    Box(Box ob) { // pass object to constructor
      width = ob.width;
      height = ob.height;
      depth = ob.depth;
    }
}

class BoxWeight extends Box {
    double weight; // weight of box

    // construct clone of an object

    BoxWeight(BoxWeight ob) { // pass object to constructor
       super(ob);
       weight = ob.weight;
    }
}
```

Notice that super() is passed an object of type BoxWeight—not of type Box.This still invokes the constructor Box(Box ob).
NOTE: A superclass variable can be used to reference any object derived from that class.
Thus, we are able to pass a BoxWeight object to the Box constructor.Of course,Box only has knowledge of its own members.

exactly same thing happening

(2)

A Second Use for super
The second form of super acts somewhat like this, except that it always refers to the superclass of the subclass in
which it is used.

super.member

Here, member can be either a method or an instance variable. This second form of super is most applicable to situations
in which member names of a subclass hide members by the same name in the superclass.

example int weight is in both Box and Boxweight
so Super.weight → access to Box weight
this.weight → access to Boxweight variable

★ Imp

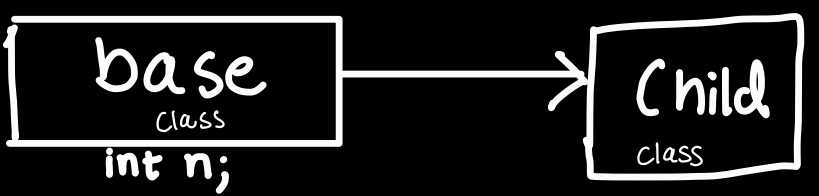
super( ) always refers to the constructor in the closest superclass. The super( ) in BoxPrice calls the constructor in
BoxWeight. The super( ) in BoxWeight calls the constructor in Box. In a class hierarchy, if a superclass constructor
requires parameters, then all subclasses must pass those parameters “up the line.” This is true whether or not a
subclass needs parameters of its own.

If you think about it, it makes sense that constructors complete their execution in order of derivation.
Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and
possibly prerequisite to any initialization performed by the subclass. Therefore, it must complete its execution first.

NOTE: If super( ) is not used in subclass' constructor, then the default or parameterless constructor of each superclass
will be executed.

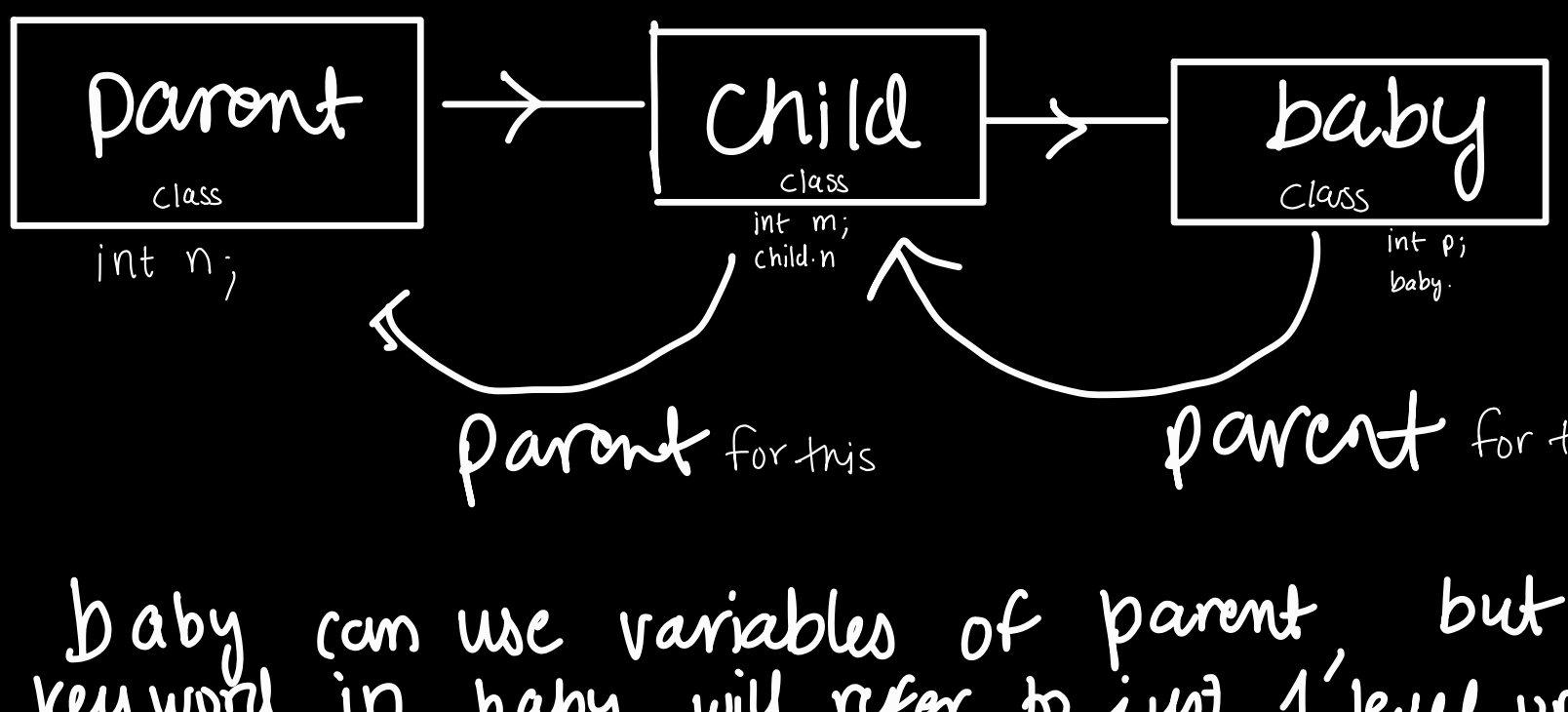
Types of Inheritance

① single



→ can use child.n

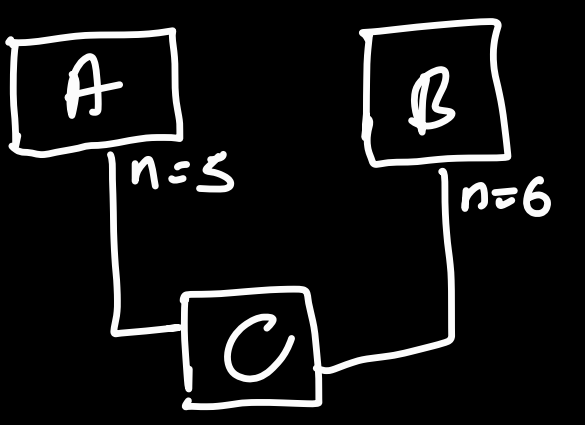
② multilevel – child class will become parent class for another baby class



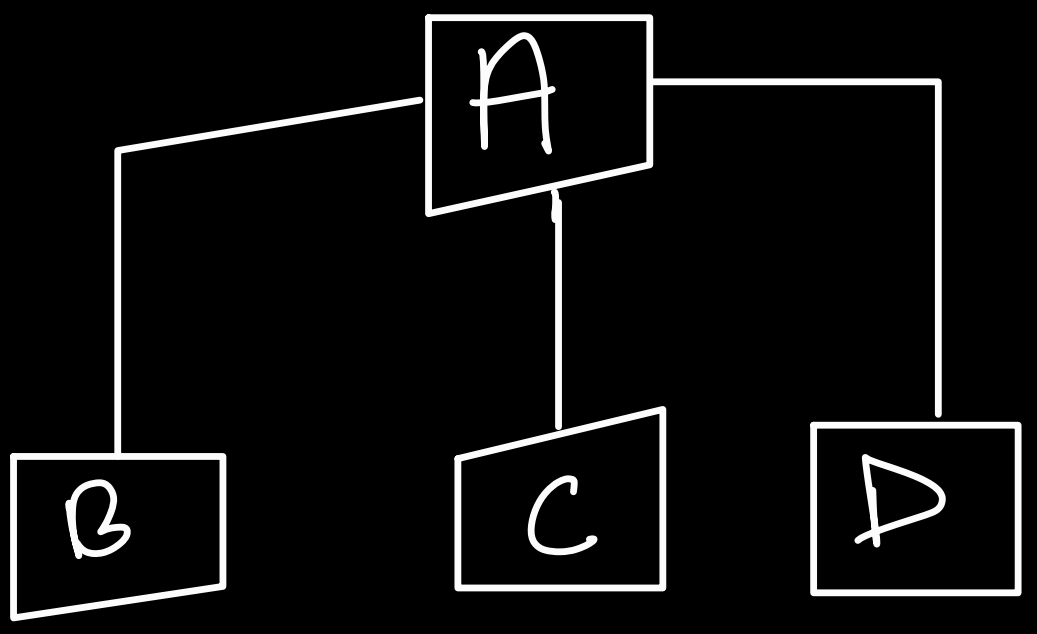
baby can use variables of parent, but super keyword in baby will refer to just 1 level up i.e Child class

③ multiple inheritance

X not supported in Java
we use interfaces.



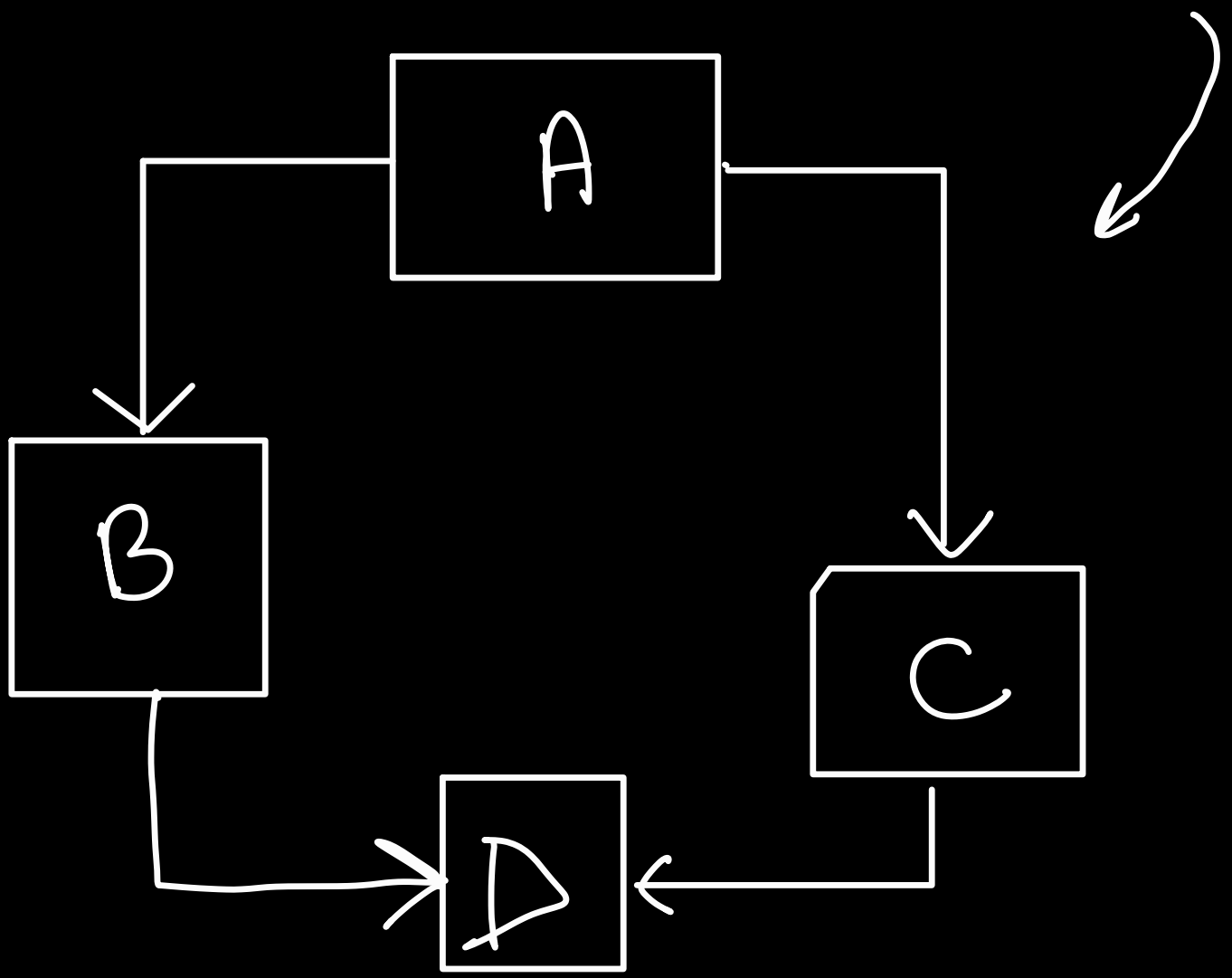
④ hejrarchial Inheritance



B, C, D can always extend A. B can't extend C.

⑤ Hybrid inheritance (not in Java)

combination of single and multiple.



Imp - A class cannot be its own super class.

Polymorphism → many → ways to represent

many ways of representing

↓ properties of OOPs.

① Compile time / static Polymorphism

Achieved via method overloading

↓

same name of the methods but type, argument, return type, ordering can be different. eg - multiple constructors

```
Class a = new Class();
```

Class b = new Class (3,4); → diff constructor

↳ Compile time polymorphism

by method overloading

Java determines which one to run at compile time.

```java
public class Numbers {

    int sum(int a, int b) {
        return a + b;
    }

    int sum(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        Numbers obj = new Numbers();
        obj.sum( a: 2,  b: 3);
        obj.sum( a: 1,  b: 3,  c: 7);
        obj.sum(4, 5, 6, 8);
    }
}
```

① → Both these methods have the same name

② → 2 parameters :. calls ①

③ → 3 parameters :. calls ②

↳ error as checking at compile time

* - if double in parameter and we give argument as int, Auto Casting happens.

② Run time / Dynamic Polymorphism

Parent has function called Area

Child also has function called Area

when object.Area is called

→ Parent object = new Child();

here, the method to be called depends on this

↓

known as upcasting

whole known as upcasting.

This decides which ones to access

This decides which one to run

how Java does this → at compile time it decides which methods it can access (∴ here from parent class) so method (Area) should be there in parent class and at compile time, the subclass's method is run, because it decides which one to run

```
In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass,
then the method in the subclass is said to override the method in the superclass. When an overridden method is called
from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the
method defined by the superclass will be hidden.

Method overriding occurs only when the names and the type signatures of the two methods are identical.
If they are not, then the two methods are simply overloaded.

(Check display functions in box classes)

Dynamic Method Dispatch:

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than
compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
Let’s begin by restating an important principle: a superclass reference variable can refer to a subclass object.
When an overridden method is called through a superclass reference, Java determines which version of that method to
execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is
made at run time.
In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

If B extends A then you can override a method in A through B with changing the return type of method to B.
```

→ A better explanation.

```java
package com.kunal.properties.polymorphism;

public class Main {
    public static void main(String[] args) {
        Shapes shape = new Shapes();
        Shapes circle = new Circle();
        Shapes square = new Square();

        circle.area();
    }
}
```

```java
package com.kunal.properties.polymorphism;

public class Shapes {
    void area() {
        System.out.println("I am in shapes");
    }
}
```

Shapes . Java

This method calls the one in area, not in super class.

This one is called at runtime instead of this

```java
package com.kunal.properties.polymorphism;

public class Circle extends Shapes{

    // this will run when obj of Circle is created
    // hence it is overriding the parent method
    @Override // this is called annotation
    void area() {
        System.out.println("Area is pi * r * r");
    }
}
```

Final Keyword to prevent Overriding →

```
# Using final to Prevent Overriding:
To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
Methods declared as final cannot be overridden.
Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them
because it "knows" they will not be overridden by a subclass. When a small final method is called, often the Java
compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus
eliminating the costly overhead associated with a method call. Inlining is an option only with final methods.
Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final
methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding.
```

Final keyword to prevent Inheritance →

```
# Using final to Prevent Inheritance:
Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.
NOTE: Declaring a class as final implicitly declares all of its methods as final, too.
As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete
by itself & relies upon its subclasses to provide complete implementations.

# NOTE: Although static methods can be inherited ,there is no point in overriding them in child classes because the
method in parent class will run always no matter from which object you call it. That is why static interface methods
cannot be inherited because these method will run from the parent interface and no matter if we were allowed to
override them, they will always run the method in parent interface.
That is why static interface method must have a body.

NOTE : Polymorphism does not apply to instance variables.
```

Static always run
∴ one in parent class will run and the same method in child class will not be overridden

overriding static methods-

∵ overriding happens from objects
∵ static doesn't deal with objects ∴

③ Encapsulation

Wrapping up the implementation of the data members and methods in a class.

④ Abstraction

Hiding unnecessary details and showing valuable information only.

Abstraction — The design part
Encapsulation — The implementation of designed part

	Abstraction	Encapsulation
1	Abstraction is a feature of OOPs that hides the **unnecessary** detail but shows the essential information.	Encapsulation is also a feature of OOPs. It hides the code and data into a **single** entity or unit so that the data can be protected from the outside world.
2	It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
3	It focuses on the **external** lookout.	It focuses on **internal** working.
4	It can be implemented using **abstract classes** and **interfaces**.	It can be implemented by using the **access modifiers** (private, public, protected).
5	It is the process of **gaining** information.	It is the process of **containing** the information.
6	In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
7	The objects are **encapsulated** that helps to perform abstraction.	The object need not to **abstract** that result in encapsulation.

achieved via

Data hiding — data security + hiding complexity (private)

Encapsulation — hide the complexity of the system only (hiding public/private)

process in data hiding

Example of encapsulation → Encryption

Example of Abstraction → using the string method

(we need not know how it plues objects data inside the class)

Object Oriented Programming IV

Access Control, In-built packages, Object Class

```
Access Control:

How a member can be accessed is determined by the access modifier attached to its declaration.
Usually, you will want to restrict access to the data members of a class—allowing access only through methods.
Also, there will be times when you will want to define methods that are private to a class.

Java's access modifiers are public, private, and protected. Java also defines a default access level.
protected applies only when inheritance is involved.

When no access modifier is used, then by default the member of a class is public within its own package,
but cannot be accessed outside of its package.

            | Class | Package | Subclass | Subclass | World
            |       |         |(same pkg)|(diff pkg)|(diff pkg & not subclass)
————————————|———————|—————————|——————————|——————————|————————————————————
public      |   +   |    +    |    +     |     +    |   +
————————————|———————|—————————|——————————|——————————|————————————————————
protected   |   +   |    +    |    +     |     +    |
————————————|———————|—————————|——————————|——————————|————————————————————
no modifier |   +   |    +    |    +     |          |
————————————|———————|—————————|——————————|——————————|————————————————————
private     |   +   |         |          |          |

+ : accessible
blank : not accessible
```

for sensitive data use by public getters setters

```
package packageOne;
public class Base
{
    protected void display(){
        System.out.println("in Base");
    }
}

package packageTwo;
public class Derived extends packageOne.Base{
    public void show(){
        new Base().display();       // this is not working
        new Derived().display();    // is working
        display();//is working
    }
}

protected allows access from subclasses and from other classes in the same package.
We can use child class to use protected member outside the package but only child class object can access it.
That's why any Derived class instance can access the protected method in Base.
The other line creates a Base instance (not a Derived instance!!).
And access to protected methods of that instance is only allowed from objects of the same package.
```

```
display();
-> allowed, because the caller, an instance of Derived has access to protected members and fields of its subclasses,
even if they're in different packages


new Derived().display();
-> allowed, because you call the method on an instance of Derived and that instance has access to the protected methods
of its subclasses

new Base().display();
-> not allowed because the caller's (the this instance) class is not defined in the same package like the Base class,
so this can't access the protected method. And it doesn't matter - as we see - that the current subclasses a class from
that package. That backdoor is closed ;)

Remember that any time talks about a subclass having an access to a superclass member, we could be talking about the
subclass inheriting the member, not simple accessing the member through a reference to an instance of the superclass.


class C
    protected member;

// in a different package

class S extends C

    obj.member; // only allowed if type of obj is S or subclass of S

The motivation is probably as following. If obj is an S, class S has sufficient knowledge of its internals,
it has the right to manipulate its members, and it can do this safely.
If obj is not an S, it's probably another subclass S2 of C, which S has no idea of.
S2 may have not even been born when S is written. For S to manipulate S2's protected internals is quite dangerous.
If this is allowed, from S2's point of view, it doesn't know who will tamper with its protected internals and how,
this makes S2 job very hard to reason about its own state.

Now if obj is D, and D extends S, is it dangerous for S to access obj.member? Not really.
How S uses member is a shared knowledge of S and all its subclasses, including D. S as the superclass has the right to
define behaviours, and D as the subclass has the obligation to accept and conform.

For easier understanding, the rule should really be simplified to require obj's (static) type to be exactly S.
After all, it's very unusual and inappropriate for subclass D to appear in S. And even if it happens,
that the static type of obj is D, our simplified rule can deal with it easily by upcasting:
((S)obj).member
```

Inbuilt pachages in Java →

- → lang - contains language specific stuff, ex- add, subtract, multiply and specific Java, no need to import this.
- → io → File reading, optimise the input output, Bufferreader
- → Util → contains data structures like ArrayList and collections framework.
- → applet
- → awt - gui
- → net - networking stuff.

Object class → Inside the lang class

→ Top most class in inheritance, every class by default extends this class.
even if multiple inheritance not allowed, but this is an exception, happens internally.

→ object class methods —
.toString, finalize, hashcode
↓
random value → algorithmic value
equals.

→ Instanceof
↳ tells whether something is a subclass of something

→ object.getClass(); → get data from a class
↳ can't override this

Object Oriented Programming V

Abstract Classes , Interfaces , Annotations

```
Sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its
subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that
the subclasses must implement.
You may have methods that must be overridden by the subclass in order for the subclass to have any meaning.
In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution
to this problem is the abstract method.
You can require that certain methods be overridden by subclasses by specifying the abstract type modifier.

        abstract type name(parameter-list);

These methods are sometimes referred to as subclass's responsibility because they have no implementation specified in
the superclass.
Thus, a subclass must override them—it cannot simply use the version defined in the superclass.

Any class that contains one or more abstract methods must also be declared abstract.
```

main.java

```java
package com.kunal.abstractDemo;

public class Main {
    public static void main(String[] args) {
        Son son = new Son();
        son.career();

        Daughter daughter = new Daughter();
        daughter.career();
    }
}
```

parent.java

```java
package com.kunal.abstractDemo;

public abstract class Parent {
    abstract void career(String name);
    abstract void partner(String name, int age);
}
```

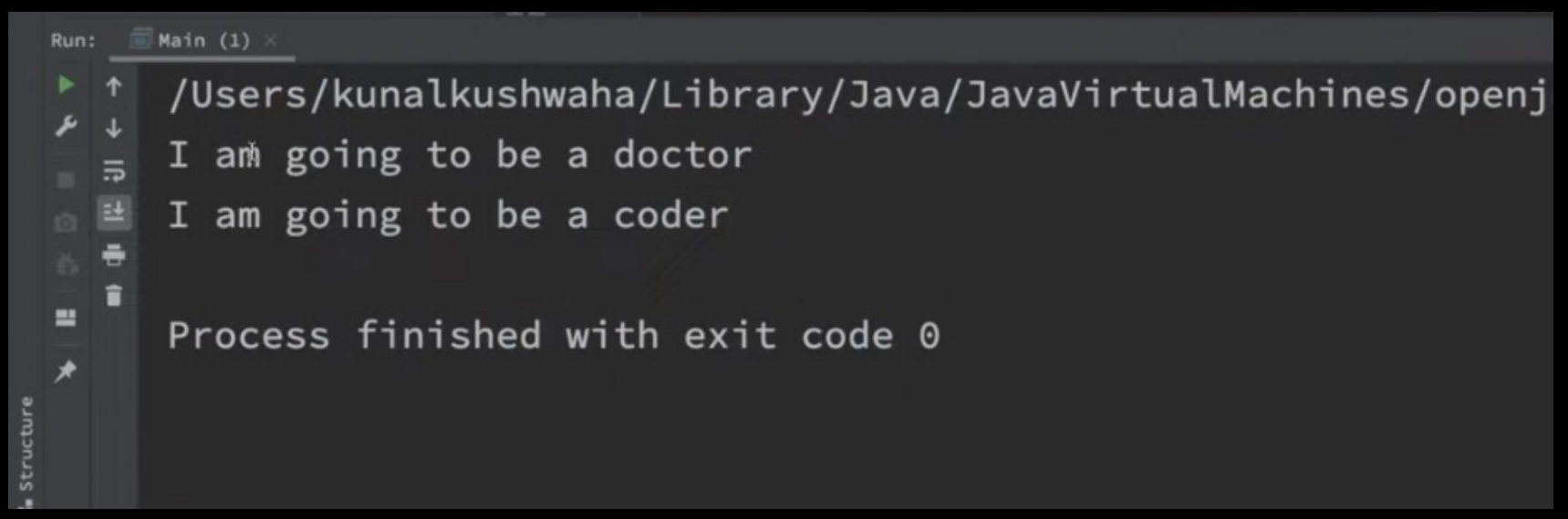
daughter.java

```java
package com.kunal.abstractDemo;

public class Daughter extends Parent{

    2 related problems
    @Override
    void career() {
        System.out.println("I am going to be a coder");
    }

    @Override
    void partner() {
        System.out.println("I love Iron Man");
    }
}
```

Son.java

```java
package com.kunal.abstractDemo;

public class Son extends Parent{

    2 related problems
    @Override
    void career() {
        System.out.println("I am going to be a doctor");
    }

    @Override
    void partner() {
        System.out.println("I love Pepper Potts");
    }
}
```

Output →



→ We cannot create objects of an abstract class.
↳ use override methods with it, coz methods inside won't have any body

→ We also cannot create constructors of abstract class + no static abstract methods

→ we can create static methods in abstract classes

```
16 # There can be no objects of an abstract class.
17 # You cannot declare abstract constructors, or abstract static methods.
18 # You can declare static methods in abstract class.
19 Because there can be no objects for abstract class. If they had allowed to call abstract static
   methods,
20 it would that mean we are calling an empty method (abstract) through classname because it is static.
21 Any subclass of an abstract class must either implement all of the abstract methods in the superclass,
22 or be declared abstract itself.
23 Abstract classes can include as much implementation as they see fit i.e.there can be concrete
   methods(methods with body)
24 in abstract class.
```

```
26 Although abstract classes cannot be used to instantiate objects, they can be used to create object
   references,
27 because Java's approach to run-time polymorphism is implemented through the use of superclass
   references.
28
29 A public constructor on an abstract class doesn't make any sense because you can't instantiate an
   abstract class directly
30 (can only instantiate through a derived type that itself is not marked as abstract)
```

We can have constructor like this → using 'super keyword' →

Yes, an abstract class can have a constructor. Consider this:

```
abstract class Product {
    int multiplyBy;
    public Product( int multiplyBy ) {
        this.multiplyBy = multiplyBy;
    }

    public int mutiply(int val) {
       return multiplyBy * val;
    }
}

class TimesTwo extends Product {
    public TimesTwo() {
        super(2);
    }
}

class TimesWhat extends Product {
    public TimesWhat(int what) {
        super(what);
    }
}
```

The superclass `Product` is abstract and has a constructor. The concrete class `TimesTwo` has a constructor that just hardcodes the value 2. The concrete class `TimesWhat` has a constructor that allows the caller to specify the value.

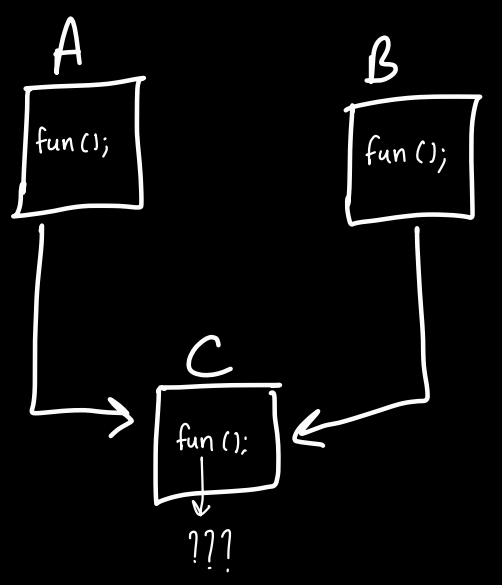
Abstract constructors will frequently be used to enforce class constraints or invariants such as the minimum fields required to setup the class.

> NOTE: As there is no default (or no-arg) constructor in the parent abstract class, the constructor used in subclass must explicitly call the parent constructor.

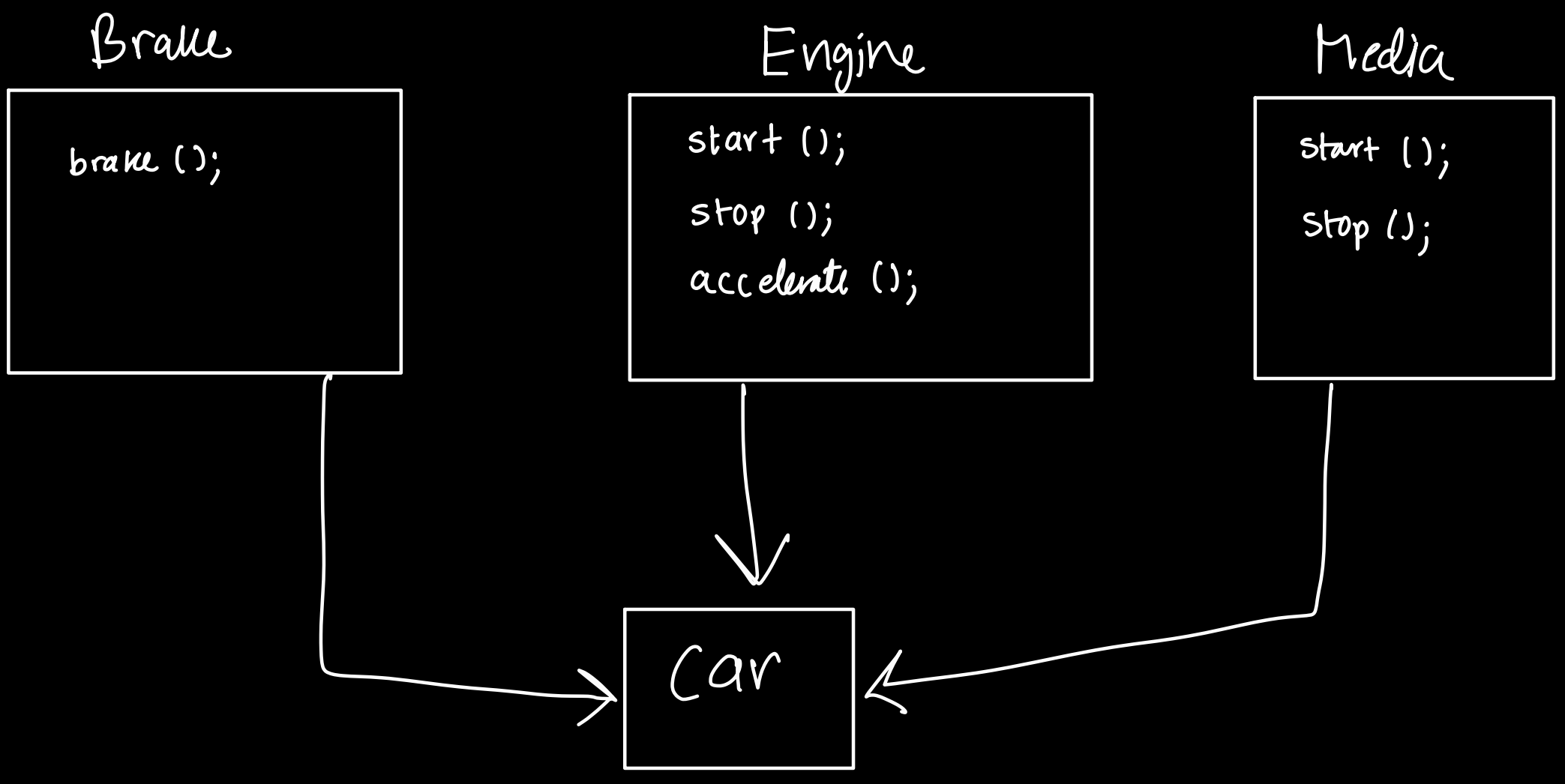
Interfaces — contains abstract classes

↳ solves multiple inheritance

→ variables in this are static and final!



Example →



This is how we will write Car.java

```java
package com.kunal.interfaces;

public class Car implements Engine, Brake{

    @Override
    public void brake() {
        System.out.println("I brake like a normal Car");
    }

    @Override
    public void start() {
        System.out.println("I start like a normal Car");
    }

    @Override
    public void stop() {
        System.out.println("I stop like a normal Car");
    }

    @Override
    public void acc() {
        System.out.println("I accelerate like a normal
    }
```

imp

This is how we will write engine.java

```java
package com.kunal.interfaces;

public interface Engine {

    static final int PRICE = 78000;

    void start();
    void stop();
    void acc();
}
```

imp

Imp Notes →

Multiple inheritance is not available in java.
(Same functions in 2 classes it will skip that hence no multiple inheritance)

Instead we have java interfaces. they have abstract functions (no body of functions)

Interface is like class but not completely. it is like an abstract class.
By default functions are public and abstract in interface.
variables are final and static by default in interface.

Interfaces specify only what the class is doing, not how it is doing it.
The problem with MULTIPLE INHERITANCE is that two classes may define different ways of doing the same thing,
and the subclass can't choose which one to pick.

Key difference between a class and an interface: a class can maintain state information
(especially through the use of instance variables), but an interface cannot.

Using interface, you can specify a set of methods that can be implemented by one or more classes.
Although they are similar to abstract classes, interfaces have an additional capability:
A class can implement more than one interface. By contrast, a class can only inherit a single superclass
(abstract or otherwise).

Using the keyword interface, you can fully abstract a class' interface from its implementation.
That is, using interface, you can specify what a class must do, but not how it does it.

Interfaces are syntactically similar to classes, but they lack instance variables, and, as a general rule,
their methods are declared without any body.

By providing the interface keyword, Java allows you to fully utilize the "one interface, multiple methods"
aspect of polymorphism.

NOTE: Interfaces are designed to support dynamic method resolution at run time.
Normally, in order for a method to be called from one class to another, both classes need to be present at compile time
so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for
a static and nonextensible classing environment. Inevitably in a system like this, functionality gets pushed up higher
and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are
designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance
hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated
in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

Beginning with JDK 8, it is possible to add a default implementation to an interface method.
Thus, it is now possible for interface to specify some behavior.However, default methods constitute what is, in essence,
a special-use feature, and the original intent behind interface still remains.

Variables can be declared inside of interface declarations.
NOTE: They are implicitly final and static, meaning they cannot be changed by the implementing class.
They must also be initialized. All methods and variables are implicitly public.

NOTE: The methods that implement an interface must be declared public. Also, the type signature of the implementing
method must match exactly the type signature specified in the interface definition.

It is both permissible and common for classes that implement interfaces to define additional members of their own.

type of variable as type of interface allowed

```
Engine car = new Car();
```

What things you access depend on this

which versions to access depend on this

if both engine and car have eg- acc variable

∴ the acc inside car will be used instead of engine acc var.

Reason

now Car class has

int a = 30;

in function Main,

we cannot access car.a;

→ wath OOP 5. 43:00 – 54:00

```
NOTE:
You can declare variables as object references that use an interface rather than a class type.
This process is similar to using a superclass reference to access a subclass object.
Any instance of any class that implements the declared interface can be referred to by such a
variable.
When you call a method through one of these references, the correct version will be called based on
the actual instance
of the interface being referred to. Called at run time by the type of object it refers to.
The method to be executed is looked up dynamically at run time, allowing classes to be created later
than the code which
calls methods on them.
The calling code can dispatch through an interface without having to know anything about the "callee."

CAUTION: Because dynamic lookup of a method at run time incurs a significant overhead when compared
with the normal
method invocation in Java, you should be careful not to use interfaces casually in performance-
critical code.

```

Now → Example

① CDPlayer.java

```java
package com.kunal.interfaces;

public class CDPlayer implements Media{

    @Override
    public void start() {
        System.out.println("Music start");
    }

    @Override
    public void stop() {
        System.out.println("Music stop");
    }
}
```

② ElectricEngine.java

```java
package com.kunal.interfaces;

public class ElecticEngine implements Engine{

    @Override
    public void start() {
        System.out.println("Electric engine start");
    }

    @Override
    public void stop() {
        System.out.println("Electric engine stop");
    }

    @Override
    public void acc() {
        System.out.println("Electric engine accelerate");
    }
}
```

③ PowerEngine.java

```java
package com.kunal.interfaces;

public class PowerEngine implements Engine{

    @Override
    public void start() {
        System.out.println("Power engine start");
    }

    @Override
    public void stop() {
        System.out.println("Power engine stop");
    }

    @Override
    public void acc() {
        System.out.println("Power engine accelerate");
    }
}
```

④ NiceCar.java

```java
package com.kunal.interfaces;

public class NiceCar {
    private Engine engine;
    private Media player = new CDPlayer();

    public NiceCar() {
        engine = new PowerEngine();
    }

    public NiceCar(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }

    public void stop() {
        engine.stop();
    }

    public void startMusic() {
        player.start();
    }

    public void stopMusic() {
        player.stop();
    }

    public void upgradeEngine() {
        this.engine = new ElecticEngine();
    }
}
```

⑤ Main.java

```java
        NiceCar car = new NiceCar();

        car.start();  → refers to power engine
        car.startMusic();
        car.upgradeEngine(); → now diff constructor used (diff parameter passed)
        car.start();
    }
}
```

Output →

```
Power engine start
Music start
Electric engine start
```

∴ output

This

Extending Interfaces →

```
Interfaces Can Be Extended:
One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes.
Any class that implements an interface must implement all methods required by that interface, including any that are
inherited from other interfaces.

```

Annotations eg - @Override, it is interface public internally

Default Methods + imp notes

```

Default Interface Methods (aka extension method) :
A primary motivation for the default method was to provide a means by which interfaces could be expanded without breaking existing code.
i.e. suppose you add another method without body in an interface. Then you will have to provide the body of that method
in all the classes that implement that interface.
Ex:
 default String getString() {
    return "Default String";
 }

For example, you might have a class that implements two interfaces.
If each of these interfaces provides default methods, then some behavior is inherited from both.
# In all cases, a class implementation takes priority over an interface default implementation.
# In cases in which a class implements two interfaces that both have the same default method, but the class does not
override that method, then an error will result.
# In cases in which one interface inherits another, with both defining a common default method, the inheriting
interface's version of the method takes precedence.
      IMP
NOTE: static interface methods are not inherited by either an implementing class or a subinterface.
i.e. static interface methods should have a body! They cannot be abstract.

NOTE : when overriding methods, the access modifier should be same or better i.e. if in Parent Class it was protected, then then overridden should be either protected or public.

```

Imp

A.Java

```java
package com.kunal.interfaces.extendDemo2;

public interface A {
    // static interface methods should always have a body
    // call via the interface name
    static void greeting() {
        System.out.println("Hey I am static method");
    }

    default void fun() {
        System.out.println("I am in A");
    }
}
```

B.Java

```java
package com.kunal.interfaces.extendDemo2;

public interface B{
    void greet();

//    default void fun() {
//        System.out.println("I am in A");
//    }

//    void fun();
}
```

Main.java

```java
package com.kunal.interfaces.extendDemo2;

public class Main implements A, B {
    @Override
    public void greet() {

    }

    public static void main(String[] args) {
        Main obj = new Main();
        A.greeting();
    }
}
```

Nested Interfaces →

A.java

```java
package com.kunal.interfaces.nested;

public class A {
    // nested interface
    public interface NestedInterface {
        boolean isOdd(int num);
    }
}

class B implements A.NestedInterface {
    @Override
    public boolean isOdd(int num) {
        return (num & 1) == 1;
    }
}
```

main.java

```java
package com.kunal.interfaces.nested;

public class Main {
    public static void main(String[] args) {
        B obj = new B();
        System.out.println(obj.isOdd(6));
    }
}
```

Notes ↓

```
Nested Interfaces:

An interface can be declared a member of a class or another interface. Such an interface
is called a member interface or a nested interface. A nested interface can be declared as public,
private, or protected.
This differs from a top-level interface, which must either be declared as public or use the default
access level.
```

```java
// This class contains a member interface.
class A {
  // this is a nested interface
  public interface NestedIF {
    boolean isNotNegative(int x);
  }
}
// B implements the nested interface.
class B implements A.NestedIF {
  public boolean isNotNegative(int x) {
    return x < 0 ? false: true;
  }
}
class NestedIFDemo {
  public static void main(String args[]) {
    // use a nested interface reference
    A.NestedIF nif = new B();
    if(nif.isNotNegative(10))
      System.out.println("10 is not negative");
    if(nif.isNotNegative(-12))
      System.out.println("this won't be displayed");
  }
}
```

Abstract Classes V/s Interface

```
33
34  Abstract class vs Interface:
35
36  Type of methods:
37  Interface can have only abstract methods.
38  Abstract class can have abstract and non-abstract methods. From Java 8, it can have default and static
    methods also.
39
40  Final Variables:
41  Variables declared in a Java interface are by default final.
42  An abstract class may contain non-final variables.
43
44  Type of variables:
45  Abstract class can have final, non-final, static and non-static variables.
46  Interface has only static and final variables.
47
48  Implementation:
49  Abstract class can provide the implementation of interface.
50  Interface can't provide the implementation of abstract class.
51
52  Inheritance vs Abstraction:
53  A Java interface can be implemented using keyword "implements"
54  and abstract class can be extended using keyword "extends".
55
56  Multiple implementation:
57  An interface can extend another Java interface only,
58  an abstract class can extend another Java class and implement multiple Java interfaces.
59
60  Accessibility of Data Members:
61  Members of a Java interface are public by default.
62  A Java abstract class can have class members like private, protected, etc.
```

Object Oriented Programming VI

Generics, Custom ArrayList, Lambda Expressions Exception handling, object Cloning.

```java
package com.kunal.generics;

import java.util.ArrayList;
import java.util.Arrays;

public class CustomArrayList {

    private int[] data;
    private static int DEFAULT_SIZE = 10;
    private int size = 0; // also working as index value

    public CustomArrayList() {
        this.data = new int[DEFAULT_SIZE];
    }

    public void add(int num) {
        if (isFull()) {
            resize();
        }
        data[size++] = num;
    }

    private void resize() {
        int[] temp = new int[data.length * 2];

        // copy the current items in the new array
        for (int i = 0; i < data.length; i++) {
            temp[i] = data[i];
        }
        data = temp;
    }

    private boolean isFull() {
        return size == data.length;
    }

    public int remove() {
        int removed = data[--size];
        return removed;
    }

    public int get(int index) {
        return data[index];
    }

    public int size() {
        return size;
    }

    public void set(int index, int value) {
        data[index] = value;
    }
```

here we have made a custom ArrayList →

→ Creates the List with default size

→ checks if List is full and then doubles it using resize

→ This doubles the data of the arrayList

→ general imp features of ArrayList

```java
    @Override
    public String toString() {
        return "CustomArrayList{" +
                "data=" + Arrays.toString(data) +
                ", size=" + size +
                '}';
    }

    public static void main(String[] args) {
//        ArrayList list = new ArrayList();
        CustomArrayList list = new CustomArrayList();
//        list.add(3);
//        list.add(5);
//        list.add(9);

        for (int i = 0; i < 14; i++) {
            list.add(2 * i);
        }

        System.out.println(list);

        ArrayList<Integer> list2 = new ArrayList<>();
//        list2.add("dfghj");
    }
}
```

→ printing the ArrayList

Problem

→ only integer arrayList can be created

solved using

called as generics

Generics →

```
ArrayList <Integer> list = new ArrayList <> ();
```

generic, can't be a primitive data type
provides type safety.

generally we add <T> in the class name like this

<T> → replaced by later → <Integer>

```java
public class CustomGenArrayList<T> {

    private Object[] data;
    private static int DEFAULT_SIZE = 10;
    private int size = 0; // also working as index value

    public CustomGenArrayList() {
        data = new Object[DEFAULT_SIZE];
    }
```

```java
    public void add(T num) {
        if (isFull()) {
            resize();
        }
        data[size++] = num;
    }

    private void resize() {
        Object[] temp = new Object[data.length * 2];

        // copy the current items in the new array
        for (int i = 0; i < data.length; i++) {
            temp[i] = data[i];
        }
        data = temp;
    }

    private boolean isFull() {
        return size == data.length;
    }

    public T remove() {
        T removed = (T)(data[--size]);
        return removed;
    }

    public T get(int index) {
        return (T)data[index];
    }
```

Array Created at Runtime

if we put T [Default_Size];

↓

will give error

because at compile time it doesn't know what T is

type is checked at compile time only

↓

Type safety would have been a problem

Java uses type erasure. → replacing T with Integer

Type erasure can be explained as the process of enforcing type constraints only at compile time and discarding the element type information at runtime.

For example:

```java
public static  <E> boolean containsElement(E [] elements, E element){
    for (E e : elements){
        if(e.equals(element)){
            return true;
        }
    }
    return false;
}
```

The compiler replaces the unbound type *E* with an actual type of *Object*:

```java
public static  boolean containsElement(Object [] elements, Object element){
    for (Object e : elements){
        if(e.equals(element)){
            return true;
        }
    }
    return false;
}
```

Therefore the compiler ensures type safety of our code and prevents runtime errors.

```java
// here T should either be Number or its subclasses
public class WildcardExample<T extends Number> {

```

wildcard example ↑ only number class allowed like String not allowed.

```java
    public void getList(List<? extends Number> list) {
        // do something
    }
```

only (Number) ↓ can now allow subclasses of number like float, etc.

↳ only Int, Double ...

Comparing Objects

main.java

```java
package com.kunal.generics.comparing;

import java.util.Arrays;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        Student kunal = new Student(12, 89.76f);
        Student rahul = new Student(5, 99.52f);
        Student arpit = new Student(2, 95.52f);
        Student karan = new Student(13, 77.52f);
        Student sachin = new Student(9, 96.52f);

        Student[] list = {kunal, rahul, arpit, karan, sachin};

        System.out.println(Arrays.toString(list));
//        Arrays.sort(list, new Comparator<Student>() {
//            @Override
//            public int compare(Student o1, Student o2) {
//                return -(int)(o1.marks - o2.marks);
//            }
//        });

        Arrays.sort(list, (o1, o2) -> -(int)(o1.marks - o2.marks));

        System.out.println(Arrays.toString(list));

//        if (kunal.compareTo(rahul) < 0) {
//            System.out.println(kunal.compareTo(rahul));
//            System.out.println("Rahul has more marks");
//        }

    }
}
```

↳ makes ascendig descending

same as this but uses Lambda expression

Student.java

```java
package com.kunal.generics.comparing;

public class Student implements Comparable<Student>{
    int rollno;
    float marks;

    public Student(int rollno, float marks) {
        this.rollno = rollno;
        this.marks = marks;
    }

    @Override
    public String toString() {
        return marks + "";
    }

    @Override
    public int compareTo(Student o) {
        System.out.println("in compareto method");
        int diff = (int)(this.marks - o.marks);

        // if diff == 0: means both are equal
        // if diff < 0: means o is bigger else o is smaller

        return diff;
    }
}
```

↑ comparing method

→ does the comparison

Lambda Functions →

```java
package com.kunal.generics;

import java.util.ArrayList;
import java.util.function.Consumer;

public class LambdaFunctions {
    public static void main(String[] args) {
        ArrayList<Integer> arr = new ArrayList<>();
        for (int i = 0; i < 5; i++) {
            arr.add(i + 1);
        }
//        arr.forEach((item) -> System.out.println(item * 2));

        Consumer<Integer> fun = (item) -> System.out.println(item * 2);
        arr.forEach(fun);

        Operation sum = (a, b) -> a + b;
        Operation prod = (a, b) -> a * b;
        Operation sub = (a, b) -> a - b;
```

→ runs a for loop

→ prints every item after multiplying it by 2

```java
        LambdaFunctions myCalculator = new LambdaFunctions();
        System.out.println(myCalculator.operate(5, 3, sum));
        System.out.println(myCalculator.operate(5, 3, prod));
        System.out.println(myCalculator.operate(5, 3, sub));
    }

    private int operate(int a, int b, Operation op) {
        return op.operation(a, b);
    }
}

interface Operation {
    int operation(int a, int b);
}
```

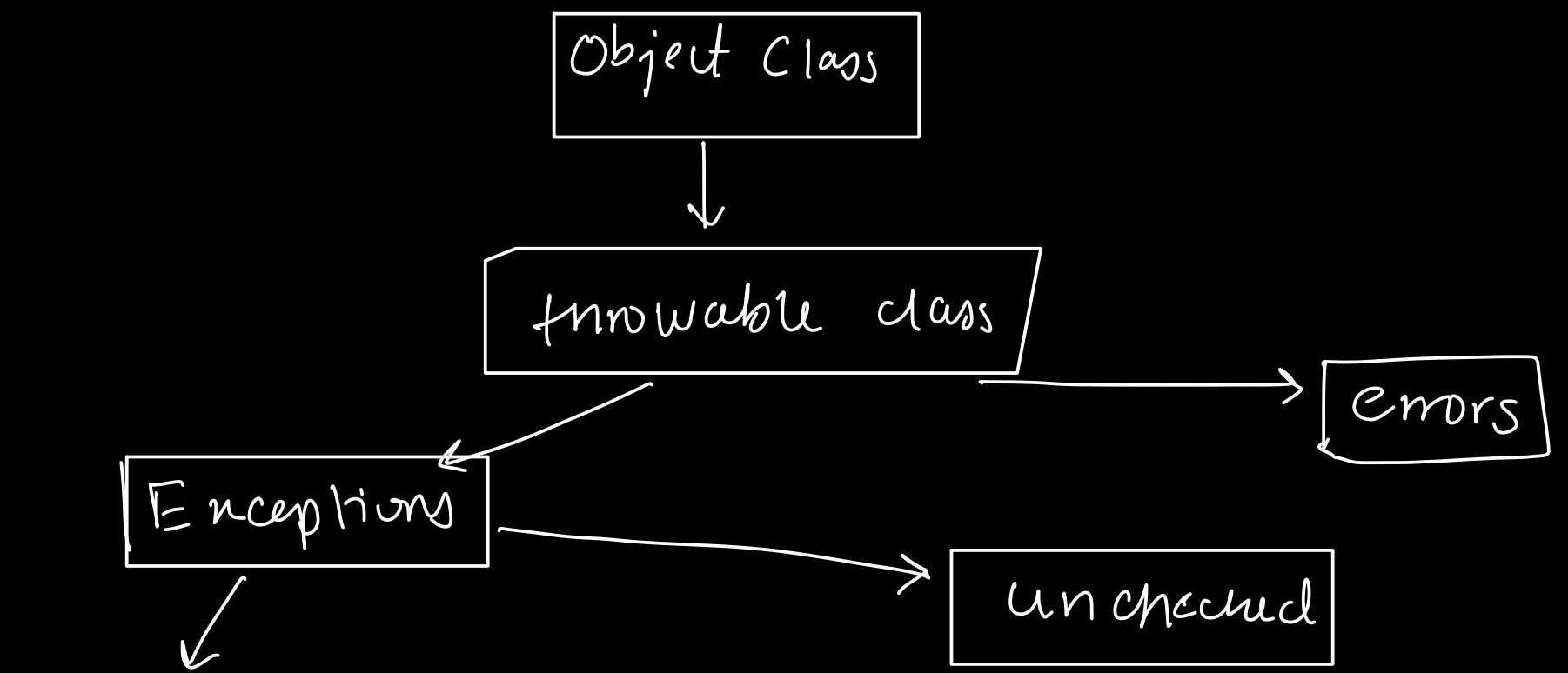
comes here ①

then here ②

③ Then goes to its body here.

Exception handeling – errors basically can't be corrected ∴ eg- stack overflow etc.

stops the flow, example dividing by zero.



Checked
happens at compile time, like filenot found.

unchecked
happens at runtime
eg- divide by zero
arithmetic exception.

myexception.java

```java
package com.kunal.exceptionHandling;

public class MyException extends Exception{

    public MyException(String message) {
        super(message);
    }
}
```

custom exception class calls constructor.

```java
package com.kunal.exceptionHandling;

public class Main {
    public static void main(String[] args) {
        int a = 5;
        int b = 0;
        try {
//            divide(a, b);
            // mimicing
            String name = "Kunal";
            if (name.equals("Kunal")) {
                throw new MyException("name is kunal");
            }
        } catch (MyException e) {
            System.out.println(e.getMessage());
        } catch (ArithmeticException e) {
            System.out.println(e.getMessage());
        } catch (Exception e) {
            System.out.println("normal exception");
        } finally {
            System.out.println("this will always execute");
        }

    }

    static int divide(int a, int b) throws ArithmeticException{
        if (b == 0) {
            throw new ArithmeticException("please do no divide by zero");
        }

        return  a / b;
    }
}
```

Pretty prints the error message (catches it and prints it)

More strict rule above (more specific) comes below

finally always executes only one finally allowed

custom exception handling

throw — means we are throwing an exception.

throws — used to declare exceptions (may throw)

Object Cloning → exact copies of an object much faster.

main.java

```java
package com.kunal.cloning;

import java.util.Arrays;

public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        Human kunal = new Human(34, "Kunal Kushwaha");
//        Human twin = new Human(kunal);

        Human twin = (Human)kunal.clone();
        System.out.println(twin.age + " " + twin.name);
        System.out.println(Arrays.toString(twin.arr));

        twin.arr[0] = 100;

        System.out.println(Arrays.toString(twin.arr));
        System.out.println(Arrays.toString(kunal.arr));
    }
}
```

anything inside function throws exception use this

simply copies the object, Slow

human.java

```java
package com.kunal.cloning;

public class Human implements Cloneable{
    int age;
    String name;
    int[] arr;

    public Human(int age, String name) {
        this.age = age;
        this.name = name;
        this.arr = new int[]{3, 4, 5, 6, 9, 1};
    }

//    @Override
//    public Object clone() throws CloneNotSupportedException{
//        // this is shallow copy
//        return super.clone();
//    }

    @Override
    public Object clone() throws CloneNotSupportedException{
        // this is deep copy
        Human twin = (Human)super.clone(); // this is actually shallow copy

        // make a deep copy
        twin.arr = new int[twin.arr.length];
        for (int i = 0; i < twin.arr.length; i++) {
            twin.arr[i] = this.arr[i];
        }
        return twin;
    }

}
```

if cannot clone

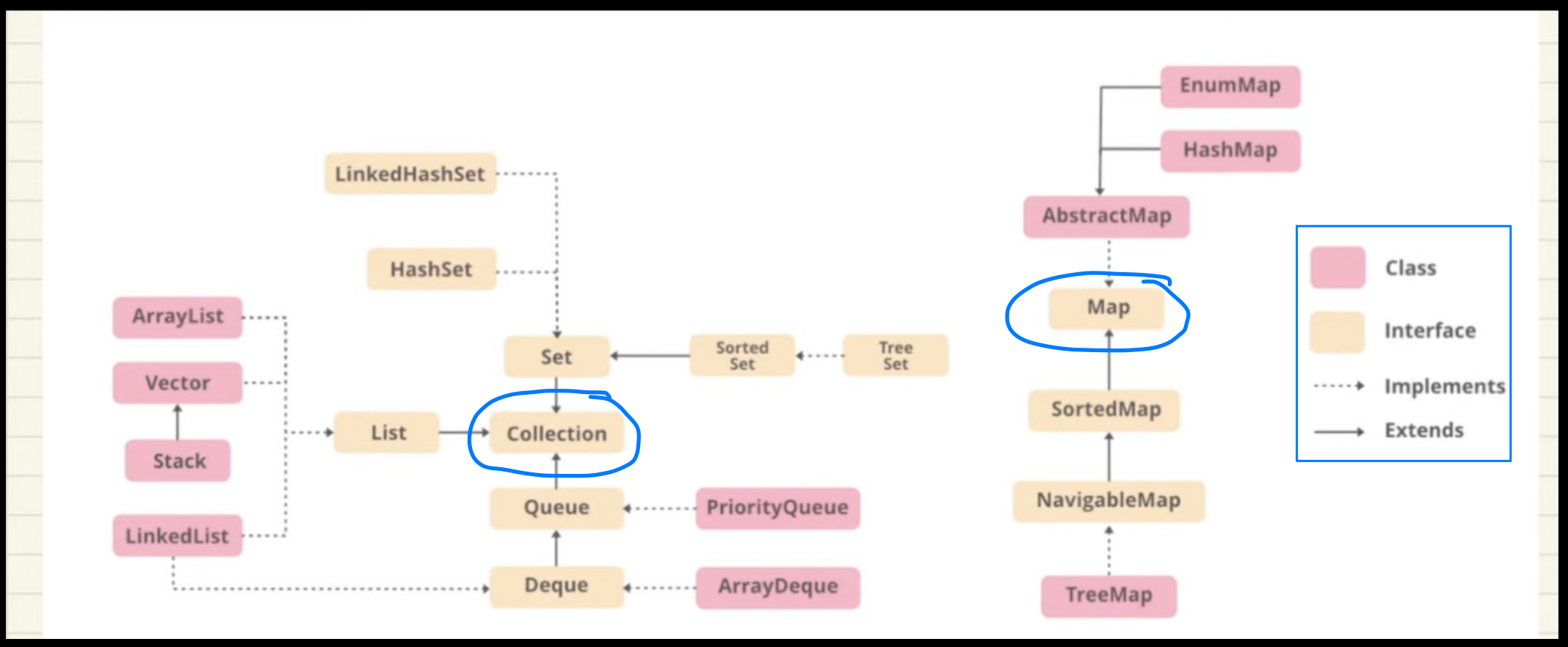
copies stuff by java internally

Shallow copy – it will copy the primitives but for arrays, it will point to the object array from which it copies

Deep copy – creates a new object (array)

Object Oriented Programming VII

Collections framework, Vector Class, Enums in Java



functions like get, remove, madd common to various data structures like linked list, stack queues etc. via abstraction is collections frame worn.

Vectors

→ multiple threads can access the ArrayList at same time.

→ single thread can access the Vector at a particular time while the othe thread waits.

Code → internally its diff, but same as array list on outside, ArrayList is fast

Enums in Java →

```
An enumeration is a list of named constants.
In Java, an enumeration defines a class type.
By making enumerations into classes, the capabilities of the enumeration are greatly expanded.

An enumeration is created using the enum keyword.
Enum declaration can be done outside a Class or inside a Class but not inside a Method
We can declare main() method inside enum. Hence we can invoke enum directly from the Command Prompt.

/* internally above enum Color is converted to (Check Example.java)
class Color
{
     public static final Color Red = new Color();
     public static final Color Blue = new Color();
     public static final Color Green = new Color();
}*/

Enum and Inheritance :
-All enums implicitly extend java.lang.Enum class. As a class can only extend one parent in Java,
so an enum cannot extend anything else.
-An enum cannot be a superclass.
-toString() method is overridden in java.lang.Enum class, which returns enum constant name.
-enum can implement many interfaces.

Two enumeration constants can be compared for equality by using the == relational operator.

values(), ordinal() and valueOf() methods :
These methods are present inside java.lang.Enum.
-values() method can be used to return all values present inside enum.
-Order is important in enums.By using ordinal() method, each enum constant index can be found,
just like array index.
-valueOf() method returns the enum constant of the specified string value, if exists.

enum and constructor :
-enum can contain constructor and it is executed separately for each enum constant at the time
of enum class loading.
-We can't create enum objects explicitly and hence we can't invoke enum constructor directly.
-And the constructor cannot be the public or protected it must have private or default modifiers.
-Why? if we create public or protected, it will allow initializing more than one objects.
-This is totally against enum concept.

enum and methods :
enum can contain concrete methods only i.e. no any abstract method.

You can compare for equality an enumeration constant with any other object by using equals( ),
which overrides the equals( ) method defined by Object.
Although equals( ) can compare an enumeration constant to any other object, those two objects
will be equal only if they both refer to the same constant,within the same enumeration.
Simply having ordinal values in common will not cause equals( ) to return true if the two constants
are from different enumerations. Remember, you can compare two enumeration references for equality by
using ==.
```

Code - Sample →

```java
public class Basic {
    enum Week {
        Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
        // these are enum constants
        // public, static and final
        // since its final you can create child enums
        // type is Week
    }

    public static void main(String[] args) {
        Week week;
        week = Week.Monday;

        for(Week day : Week.values()) {
            System.out.println(day);
        }
    }
```

```java
    Week() {
        System.out.println("Constructor called for " + this);
    }
    // this is not public or protected, only private or default
    // why? we dont want to create new objects
    // this is not the enum concept, thats why
```

↳ output of this constructor was all days printed