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

Student 1

28 Rahul 92.7

Student 2

33 Anais 95.38

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).

(4)

```
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 →

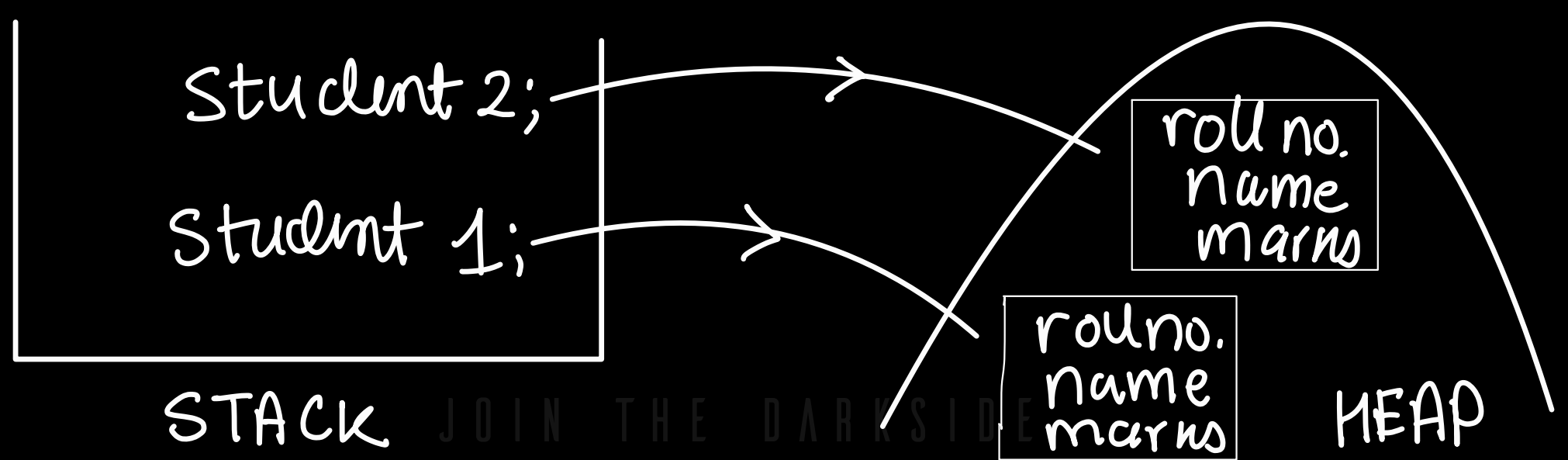
```
Student student1;          // declaring variable

student1 = new student();
```

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

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

Comile 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.rno = 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;
```

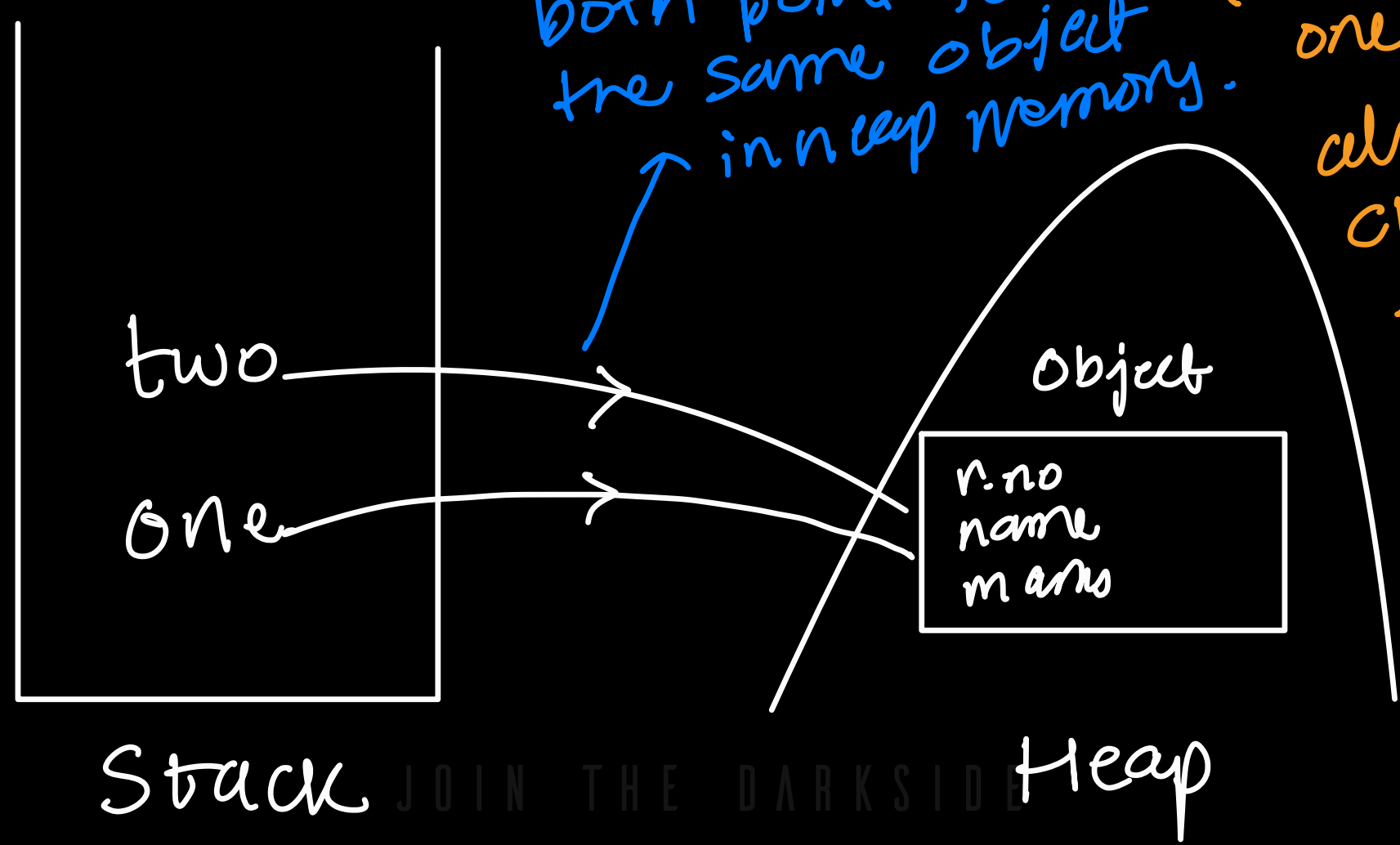
calls this constructor.

☆

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

(change in one will also change for two)

both point to the same object in heap memory.



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 convert Int to object

∴ now we can use

num . xxx = yyy;

↳ 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 vr.

eg-

final Student kunal = new Student();

kunal.name = "new name";

this is allowed

✗ kunal = other object;

this is not possible, can't be reassigned.

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

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

```
    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
}
```