Object Oriented Programming (OOPs)

Classes

A class is a named group of properties and functions. It creates a new datatype that can be used to create objects.

Ex. Say your teacher asks you to store data of 5 students using a datatype. [include roll no, marks, name]

- what you will do at first is create three different arrays. But that is not it. We have a simpler approach. Here is where Class comes into play.

you can combine the properties ex. roll no, marks, name in one single entity.

- You can actually make your own datatype. you can do it using classes.

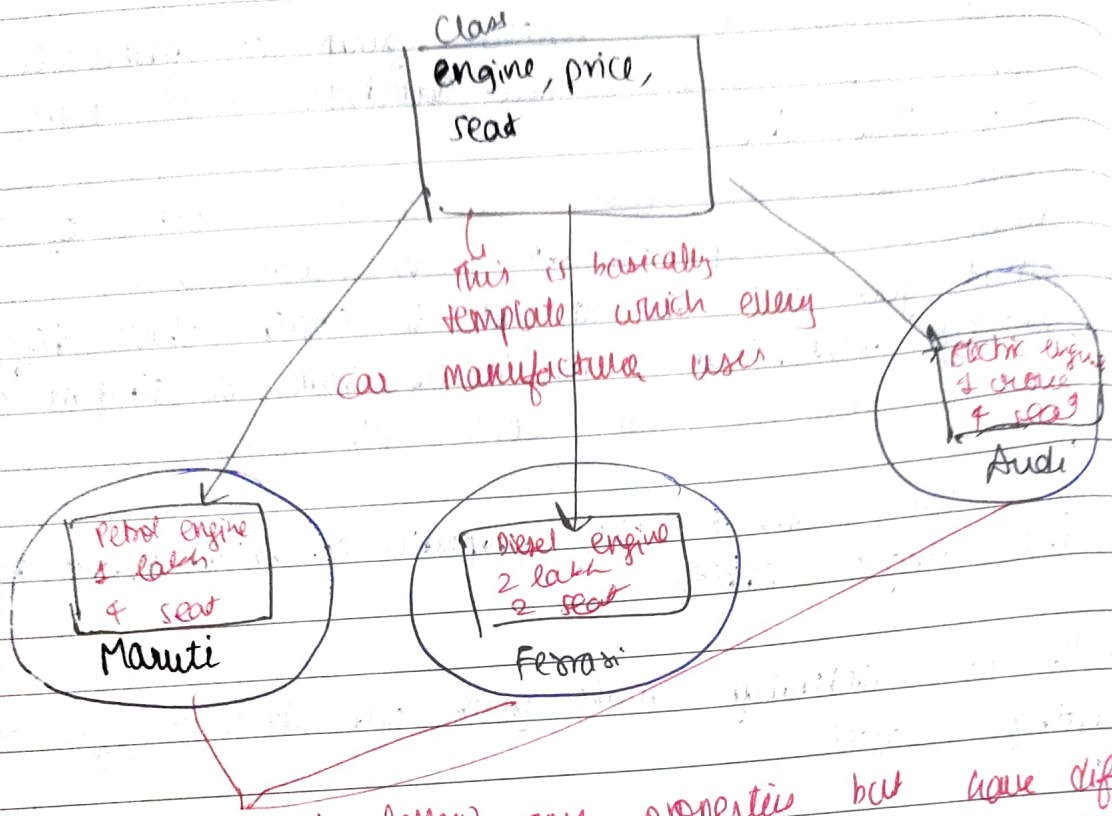
```
//create a class
class Student {
    int rno;
    String name;
    float marks;
}

psvm {
    int rno[] = new int[5];
    Student[] students = new Student[5];
    Student runal; ← (not correct, just for understanding)
}
```

- These were just properties, but you can add functions as well.
- ~~Real life example of classes:~~ Cars

(P.T.O)

```
String[] name = new String[5];
float[] marks = new float[5];
```

Ex- Real-life example : Here lets say car is class.



They all follow same properties but have different values - say difference in engine, seat, price, etc.

Property same — engine

Value — different.

Objects.

- Think, does the template stuff exist physically? No, right, but the other three do exists physically, like actual cars. The template stuff was just the rules sort of.

~~Object~~

- So, the "blue-circled thing are called **Objects**.

→ A class is template of an object.

* An object is an instance of class.

Occupies space in memory

↓

physical stuff.

Ex- New born are instance of human.

⇒ Class
Class
Object

⇒ When an const

⇒ Obj

• Th
• Tw
ar
a
• T
h

objects

⇒ Class vs Objects

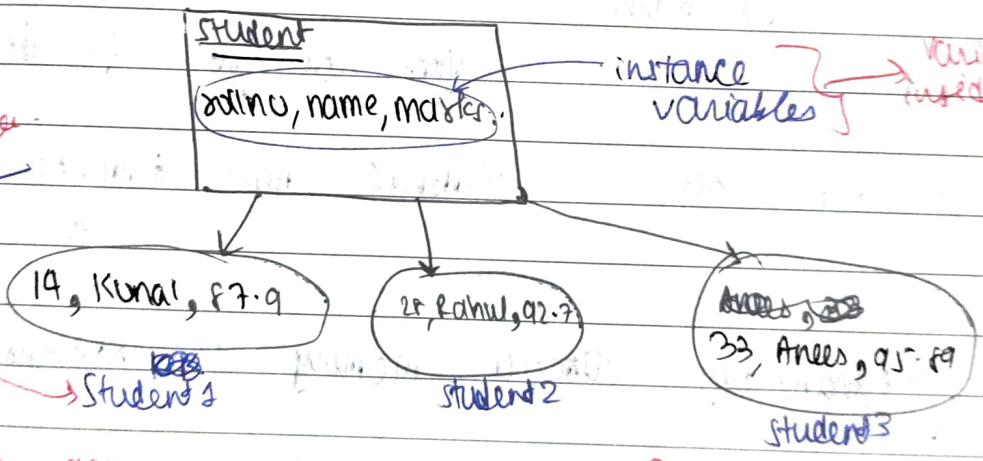
Class is a logical construct, doesn't exist physically
Object is physical reality || actually occupies space in memory.

⇒ When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct.

⇒ 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.

Take following example and assume object is already created.



⇒ How to access instance variables?

Ex What is Student1's rollno, name, etc.? What is Student2's name? etc
How to access these?

- Here, we use dot operator.

- operator: sout(student1.rollno)

links name of reference variable with instance variable.

- 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.

⇒ How to create Object?

The 'new' keyword dynamically allocates (that is, allocates at run time) memory of 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.

Ex:

```
Class Student{
    int no;
    String name;
    float marks;
}
```

To create object.

Student student1; //declare reference to object.

At this point, student1 does not yet refer to actual object.

student1 = new Student(); //allocate a Student object.

dynamically allocates memory (at runtime) & returns reference to it.

⇒ Dynamic Memory Allocation.

Student student1 = new Student();

Student student1; → compile time

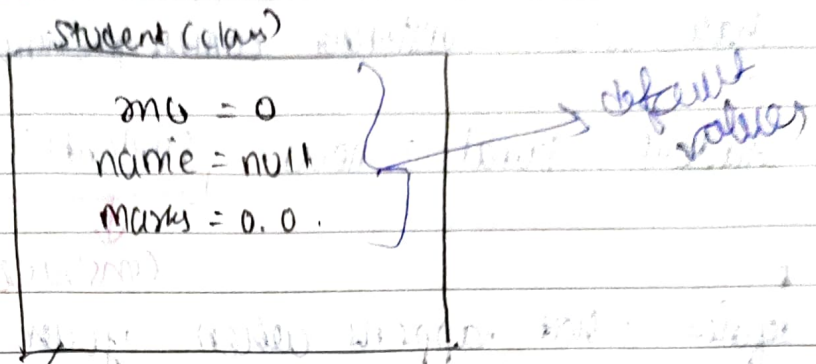
new Student(); → Run time.

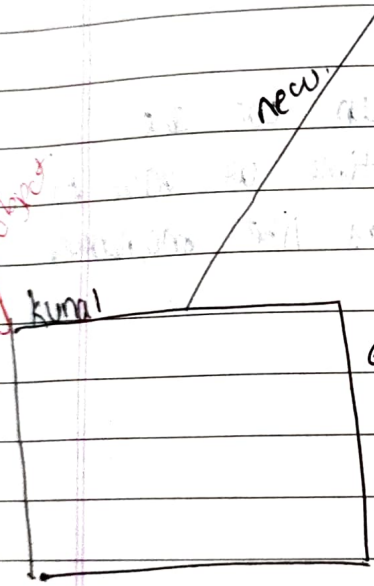
"new" does dynamic memory allocation.

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

16/7/23

JAVA Constructors





kunal.rno → if you do in java.

① It checks here if it exists, it does not it looks at default value.

② in class, rno=name 0

Similarly for kunal.marks & kunal.name

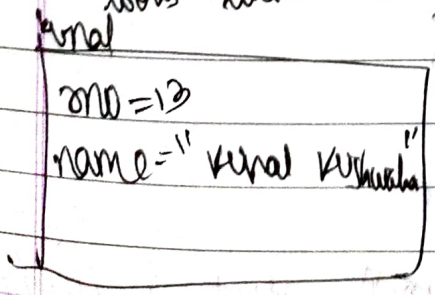
BUT, when you do.

kunal.rno = 13. → it means rno inside kunal's object.

kunal.name = "Kunal Kushwaha"

Now when you print, kunal.rno it prints 13.

Now it looks like



Now, say there were 100 objects then, would you write:

```
Obj1.no = 13;
Obj2.no = 14;
```

for 100 times ?

No, right.

That when constructor comes, ~~previous~~

Ex Student kunal = new Student();

Constructor — It basically defines what happens when your object will be created.

→ What do we want ? We want that when we are creating an object, during that time we have to add the values of all the variables like rno, name, marks that a class have.

= Instead of doing,

```
Student Student1 = new Student();
Student1.rno = 13;
Student1.name = "Kunal";
Student2.mark = 40;
```

= Using constructor, we can do,

```
Student Student1 =
        new Student (13, "Kunal", 40);
```

→ How does this works internally ?

Constructor is a special func" that is inside a class. [It runs when you create an object and it allocates some variables]

But we didn't specify any method, then how can we say this Student() is a constructor.
This is called By Default Constructor.

When you don't have any constructor inside the class, because then by default constructors comes into play.

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

To create constructor,

```
class Student {
    int rno;
    String name;
    float marks = 90;
    Student() {
        this.rno = 13;
        this.name = "Kunal";
        this.marks = 88.5f;
    }
}
```

Creates Constructor

It runs when you create a new obj.

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

// we need one word to access every object.

this keyword.

O/P:

```
13
Kunal
88.5
```

⇒ You can add func in class:

```
class Student {
    int rno;
    String name;
    float marks = 90;
    void greeting() {
        sout("Hello my name is " + name);
    }
    Student {
        this.rno = 13;
        this.name = "Kunal Kushwaha";
        this.marks = 88.5f;
    }
}
```

In main you write,

```
kunal.greeting();
```

It prints,

Hello my name is Kunal Kushwaha.

this keyword

In Java, this is a reference variable that refers to the current object.

⇒ this: refer to a current class instance variable.

Ex • Let's take a case when we do not use this keyword then what happens actually.

```
class Main {
    psvm {
        Student s1 = new Student (111, "ankit", 5000f);
        s1.display();
    }
}
class Student {
    int rollno;
    String name;
    float fee;
    Student (int rollno, String name, float fee) {
        rollno = rollno;
        name = name;
        fee = fee;
    }
    void display() {
        sout(rollno + "  " + name + "  " + fee);
    }
}
```

Op: 0 null 0.0

= now you can notice that the values of roll no, name and fee do not change even we have assigned the values in the constructor. This is because the compiler is not able to distinguish local variables and instance variables because of the

same names. This issue gets resolved using this keyword

Eg:
```
class Main{
    psvm {
        Student s1 = new Student(50, "vinay", 55000);
        s1.display();
    }
}

class Student {
    int rollno;
    String name;
    float fee;
    Student(int rollno, String name, float fee){
        this.rollno = rollno;
        this.name = name;
        this.fee = fee;
    }
    void display(){
        sout(rollno+" "+name+" "+fee);
    }
}
```

O/P: 50 vinay 55000

⇒ Why we don't use new keyword for creating primitive datatype?

Bc in java, primitive datatypes are not implemented as objects.

Objects stored in heap memory, primitives in stack memory.

Student one = new Student();

Wrapper class

int a = 10; → primitive, not much functions can be performed on it.

Integer num = 45;
// wrapper class

⇒ 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 common coding convention to choose all uppercase identifier 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.

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 of an object.

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