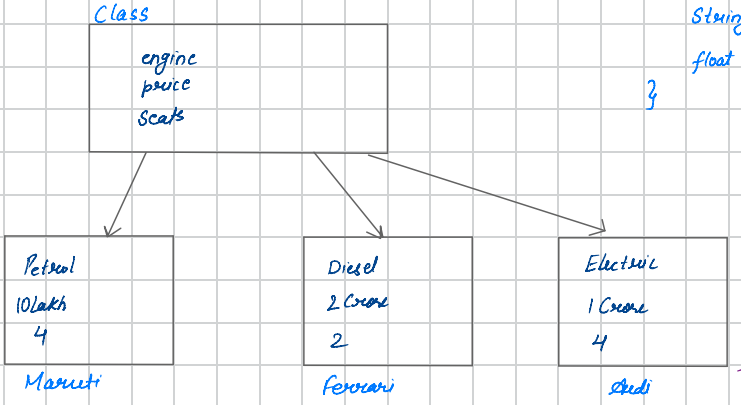

G
O
D
O
O

→ Class:→ a named group of properties & functions

↳ by convention → class starts with capital letters

```
class Student {
    int    rno  = new int   ;
    String name = new String ;
    float  marks = new float ;
}
```



→ instantiated → when you create object of an class

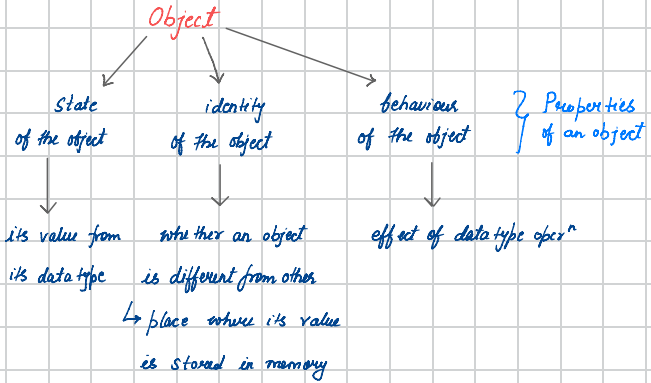
⟹ Object :→ instance of a class

↳ physical reality → occupy space in memory

Class :→ template of an object

↳ logical construct

↳ a class creates a new data type that can be used to create objects.



→ . (dot) operation :→ links the name of an object with the name of an instance variables.

↳ variables inside object

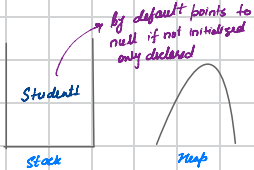
↳ Student1.rno

→ Student student1; // declare ⟹ not points to anything

by default points to null

this null is not in Heap

it is recognised by JVM



→ "new" keyword → dynamically allocates memory for an object & returns a reference to it

(allocates at run time)

↳ this reference is then stored in the variable

↳ In Java, all class objects must be dynamically allocated

↳ this reference is like the address in memory of the object allocated by "new"

Student Student1 = new Student ();

compile time — runtime (JVM)

(by default constructor)

→ Java Constructor: → Special type of function in class that runs when you create an object & it allocates some variable.

↳ it defines what occurs when an object of a class is created

↳ have no return type, not even void

↳ because implicit return type of a class constructor is the class type itself

→ Student kunal = new Student (13, "kunal kushwaha", 843);

(not a by default constructor now)

(N) → Java primitives types are not implemented as "objects"

Rather they are implemented as "normal variable"

This is done in the interest of efficiency

In Java, primitives are passed by value

not by reference

→ "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.

→ 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.

```
int square(int i){
  return i*i;
}
```

→ A parameter is a variable defined by a method that receives a value when the method is called.

For example,

in square(int i), i is a parameter. An argument is a value that is passed to a method when it is invoked.

For example, square(100) passes 100 as an argument. Inside square( ), the parameter i receives that value.

→

```
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.

(N)

"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;
```

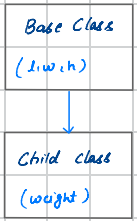
Unfortunately, final guarantees immutability only when instance variables are primitive types, not reference types.
If an instance variable of a reference type has the final modifier, the value of that instance variable (the reference to an object) will never change
—it will always refer to the same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

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
}
```

Inheritance :-



Child class is inheriting properties from base class & also have its own properties

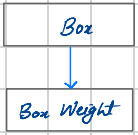
```
class child extends Base {
    int weight;
}
```

```
Child child = new Child()
child.l
child.w
child.h
```

↳ initialize parent class variables also whenever you call constructor like this

→ Types of Inheritance:-

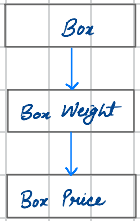
1. Single Inheritance: One class extends another class

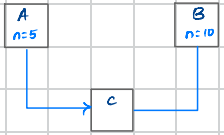


2. Multi-Level Inheritance: one class extending to more than one class

↳ Not allowed in Java

(will do in interfaces)



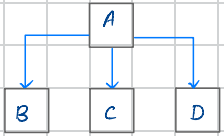


C obj = new C();

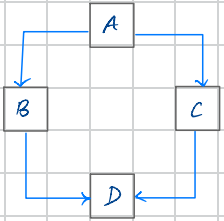
C.n → ? → Problem → not able to decide which one to pick

⇒ This why Java does not supports this

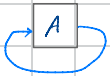
3. Hierarchical Interface: one class is inherited by many classes



4. Hybrid Inheritance : combination of single & multiple inheritance
↳ (Not in Java)
(will do in Interface)



5. Cyclic Inheritance :- Not Allowed → a class cannot be own superclass



* Polymorphism :-
many ways to represent — act of representing same thing in multiple ways

→ many ways to represent single entity or item

→ Types of Polymorphism :-

1. Compile Time / Static Polymorphism :-
↳ achieved by "Method Overloading"
↳ same name but types, arguments, return types, ordering can be different
↳ order of type → in ( )

eg:→ Multiple Constructors

```
A a1 = new A();
A a2 = new A(3,4);
```

2. Runtime / Dynamic Polymorphism :-

↳ achieved by "Method overriding"

→ Parent obj = new child();

here, which method will be called depends on type of object or this thing

This is known as Upcasting

How ⇒ Overriding works...

Object type → defines which one to run

Reference type → defines which one to access

→ How Java determines this ?

→ Dynamic Method Dispatch

↳ mechanism by which a call to an overridden method is resolved at run time rather than compile time

→ Can we override static methods ?

→ Overriding depends on objects

Static does not depend on objects

Hence, static methods cannot be overridden.

* Encapsulation :-

↳ wrapping up the implementation of the data members & methods in a class.

* Abstraction :-

↳ Hiding unnecessary details & showing valuable information.

(N):- Data Hiding is achieved by Encapsulation

Encapsulation is subprocess of Data Hiding

Abstraction	Encapsulation
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.
It solves an issue at the **design** level.	Encapsulation solves an issue at **implementation** level.
It focuses on the **external** lookout.	It focuses on **internal** working.
It can be implemented using **abstract classes** and **interfaces**.	It can be implemented by using the **access modifiers** (private, public, protected).
It is the process of **gaining** information.	It is the process of **containing** the information.
In abstraction, we use **abstract classes** and **interfaces** to hide the code complexities.	We use the **getters** and **setters** methods to hide the data.
The objects are **encapsulated** that helps to perform abstraction.	The object need not to **abstract** that result in encapsulation.