Object-Oriented Programming

#LEC-1

Java Classes:

- what is a class?

→ In short, A class is a named of a group of properties and functions.

> let's understand it with an example:

- Suppose your teacher told you to create a data type that store roll numbers of 5 students

// You may do it like this
// store 5 roll nos

```
int [] numbers = new int [5];
```

// You have created an arr and then store it

- Now your teacher told you to store name of 5 students by creating a datat structure. Again

// Again, You might created an arr of string type and then store it.

```
String[] names = new String[5];
```

- Now, Your teacher asked you to create some sort of a data structure, which can store data of 5 student. And this data structure includes - Roll.No, Marks and Name of each individual students.

Naive way :

First, you may create the integer type array to store the roll. no data of 5 students
Then, you may create the String type array to store the Names of 5 student,
Then again, you will create the float type array to store the marks of 5 students

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

✗ this is wrong approach

This is wrong as you have created different different data type for every single property.

- But we want a data structure, where every single element contains, all these three properties all together.

Using Java Classes :

We can combined above three quantities using classes. * class start with a capital letter *.

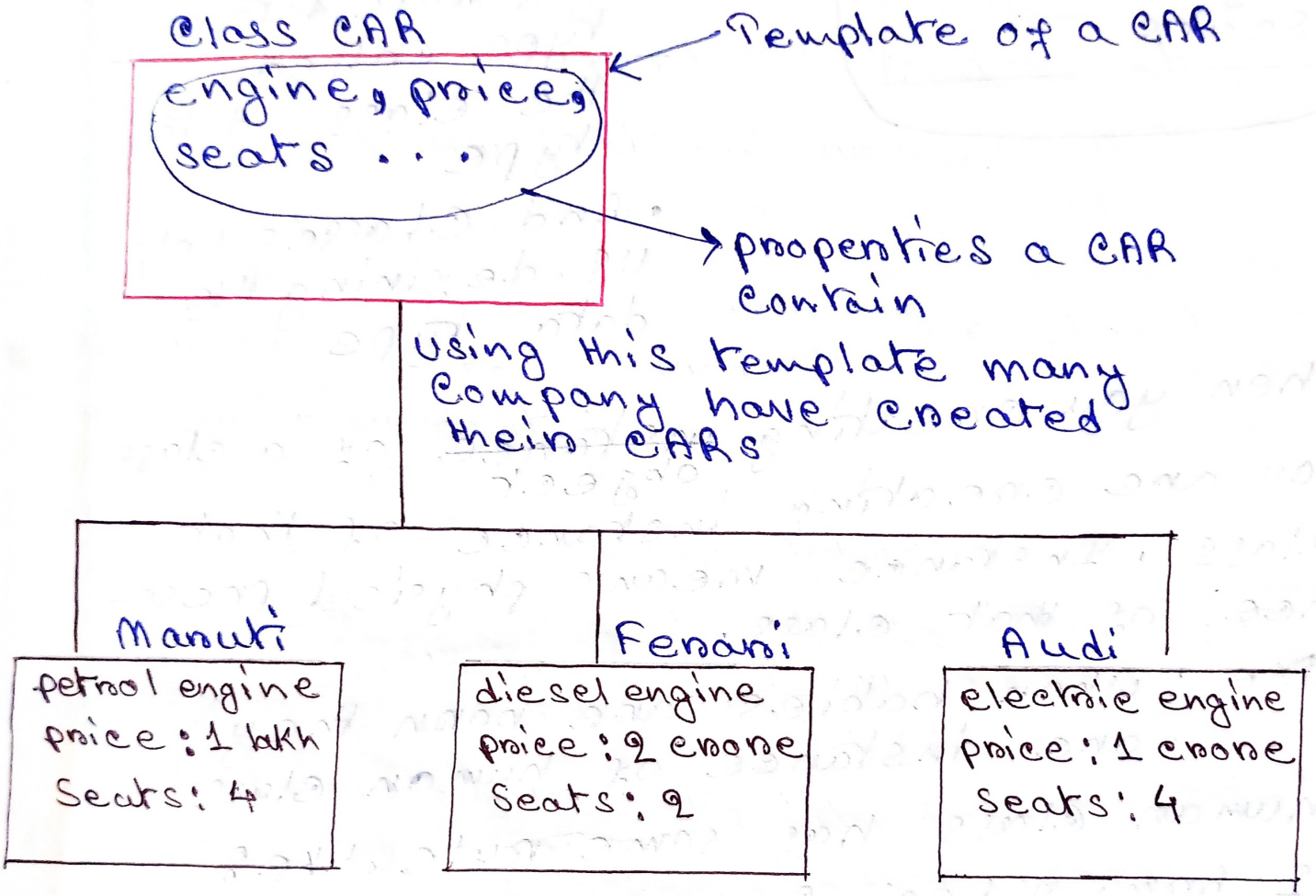
If you want to create your own data type you can do it using classes.

```
class Student {
    int rollno;
    String name;
    float marks;
}
```

Class Name

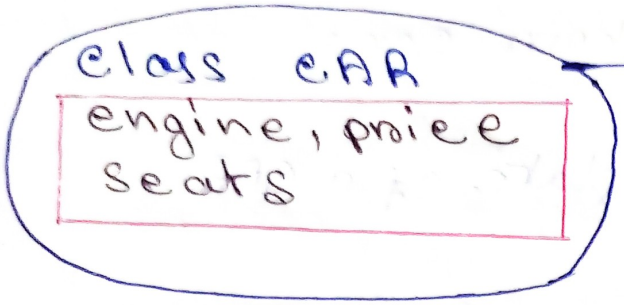
// ~~How~~ ~~Suppose~~

> A real life example of Classes is: CAR and using this cars template so many companies creating their cars



- we can see all the type of cars have same properties, but the value of the properties are different
- The class CAR or the template of a CAR doesn't exist physically, it's an idea or logic, which companie's use to produce cars, like: Maruti, Ferari, Audi etc. → which are objects.

> So, A class is a template of an object
> and an object is a instance of a class



- This can be a separate data type called the class data type
- And classes help us defining this data type

> When you creating ~~instance~~ object of a class you are creating instance of that class. Instance means physical presence of that class.
(like: when babies are born they are instance of human class)

As, human class has some rule, like 2 Arm, 2 Leg, 2 eye, 1 nose and so on. In order to make a physical thing human have to make babies.

So, babies are instance of a human class

same goes for cars there is rule for cars, engine, price, seats etc should be there

And inorder to make it as physical stuff, we have to follow this class CAR

template to create ~~object or~~ instance out of it, that is known as object

class → logical construct

Object → physical reality (This is the thing which actually occupying spaces, physically in memory)

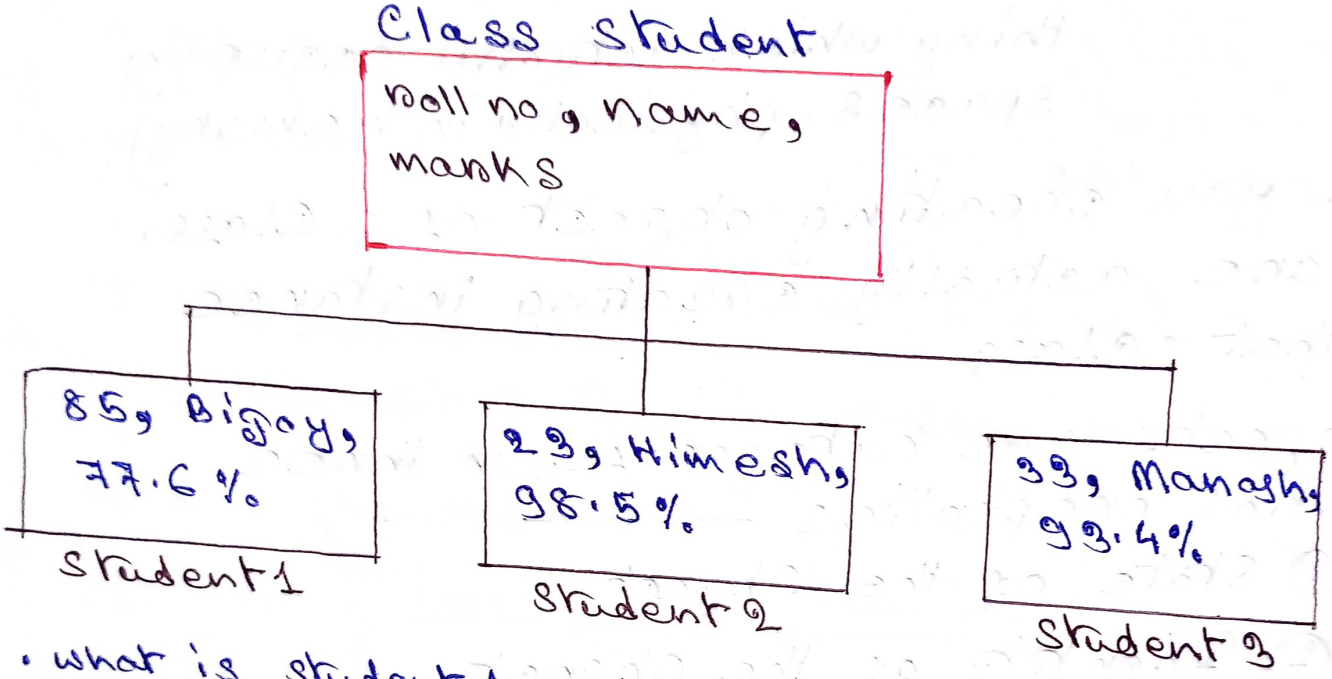
- When you creating object of a class, you are actually creating instance of that class

- The objects are categorize in three essential properties
  1. State of the Object
  2. Identity of the Object
  3. Behaviour of the Object

1. state of the object means value from it's data type
2. Identity of the object means weather one object is different from other
   - useful to things, ~~where~~ identity is like where the values store in memory. ~~teh~~ weather it's in Stack memory or Heap memory.
3. Behaviour of the object is the effect of the data type operations
   - like we will be creating functions in the objects as well. like, example - every human should have a function called

greeting. So every human will greet someone when you call the function greeting.

Now, suppose we have a student class



- what is student 1 roll no., what is student 2 percentage, what is student 3's name (How do we actually access this things?)

→ we know, classes are nothing but data types, and we can access these data types inside object using dot (.) operator.

> dot operator links the reference variable of the object with the name of the instance variable

> variables inside the object known as instance variable

So, the dot operator going to link this (roll no) with (student 1)

- So, when we do : sout (Student1.rollno) we will going to get → 85 (as output)
  - > so, we can access these instance variable using dot operator. (variables that are declared inside the class)

■ How to create an object ?

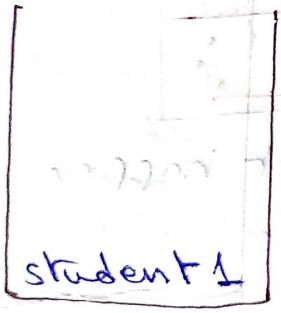
```
class Student{
  int roll_no;
  String name;
  float marks;}
```

// first we have to create a class type. As we have created student type class here

↓

```
Student student1;
```

// then we have to declare the reference variable (student1) to an object of Student type.



// At this point student1 doesn't refer to any actual object, it is just in the stack memory.

So, student1 is in stack memory we are not defining any object using it then. what it is pointing to ?
→ It is pointing to Null.

So, when the Student is not initialize by default it's Null in Java for objects.

Now, if we want create an object of this (student 1)

```
student 1 = new Student();
```

// `new` operator dynamically allocates the memory at run time, and returns a reference variable to it, which gonna be stored in student 1

* Hence, All Class objects in Java must be allocated dynamically *

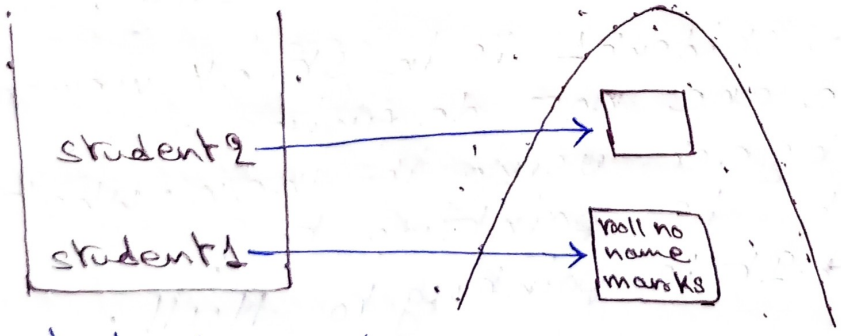
■ Dynamic Memory Allocation :-

So, when you write something like :

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

This L.H.S part happen at Compile Time

This R.H.S part happen at Run Time



➤ So, student 1, student 1 are the reference variables pointing to the actual objects that are in the heap. (we can not access the memory address, because

in Java it's not allowed).
So, this is the key for Java safety, that
* you can not manipulate references *
as you can in C, C++ (where we have concept of pointers)

- So, you can not cause an object reference to point to an arbitary memory location or manipulate it.

- new → It does dynamic memory allocation.

■ How to manipulate objects?

```
Student Kunal;  // we have declared the r.v
Kunal = new Student();  // and we have initialize the r.v
```

→ we can write this two line in one line also,

```
Student Kunal = new Student();
```

- Now, if we print the 'Kunal' [sout(Kunal)] → It will give us random value or some arbitary value.
- But, if we print [sout(Kunal.rollno)] → By default it will give us zero ('0').
- If, we print [sout(Kunal.name)] → By default it, will give us

* So, in conclusion we can say • Primitives has some default values. but • Printing objects gives us arbitary values

because, these are the more complex data type.*

So, I can modify these data types according to my needs as well.

```
Kunal.rollno = 13;  // accessing Kunal.rollno. and fixed it to 13
Kunal.name = "Kunal Kushwaha";
Kunal.marks = 88.5f;
                    // Now if we print all of these

Sout(Kunal.rollno);
Sout(Kunal.name);
Sout(Kunal.marks);
```

Output: we get → 13
Kunal Kushwaha
88.5

- Now, let's say we didn't define Kunal.marks is equal to something and we have it set some default value to the class Student

```
class Student {
    int rollno;
    String name;
    float marks = 90;
}

Sout(Kunal.marks); // After printing we get
```

Output: 90.0

- So, Why this is happening? How it's happening?

→ basically what happens is, when you create a class with bunch of properties.

```
class Student{
int rollno;    → = 0
String name;   → = null
float marks;   → = 0.0
}
```

these are default values.

and then you create an object of type Student

```
Student student1 = new Student;
```

And Now if you print it like this ⬇

```
sout(student1.rollno);
```

It's gonna print the default value 'zero'

- first it's gonna check in the object if it's define there or not,
- If it's not defined it's gonna print the default value of that data type of that particular property.

Now, Suppose the values are being declared, like

```
student1.rollno = 73;
student1.name = "Rohit";
student1.marks = 87.5f;
```

and Now if we print it.

```
sout (student1.rollno);
sout (student1.name);
sout (student1.marks);
```

we gonna get Output like: 79
Rohit
87.5

* As these values are already defined in the object of the of the refference variable *

But, you can not do.

```
sout (student1.salary);
```

// because salary doesn't defined inthe class Student

[Java is a static type language you can not do it].

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

This thing is Construcion.

- Construcion: A Construcion basically defines what happens when your obgeet will be created. It's a special type of function.

After Construcion is being created we need to allocate these

```
student1.rollno = 79;
student1.name   = "Rohit";
student1.marks  = 87.5f;
```

let's see how to write those using a constructor.

```
Student student1 = new Student(73,"Rohit",87.5);
```

→ Special type of function inside a class, that run when you create an object and it allocates some variables, as you like it

- A function must have an argument but the constructor by default doesn't have any arguments, this is known as by default constructor.

Now,

```
class Student{
    int rollno;
    String name;
    float marks;
    Student(){
    }
}
```

→ This is a constructor & the return type is the class itself

Student

→ This going to create a type of the object of the class Student hence the return type is student itself. * Name is not required *

Now, we need one word to access every object.

→ and we can do that using 'this' keyword

- whenever you try to access any particular item of the class via it's object in order to do that we use 'this' variable

```
Student student1 = new Student();
Student student1 = new Student(73, "Rohit", 87.5f);
class Student{
int rollno;
String name;
float marks;
void greeting(){
sout("Hello! My name is" + this.name);
}// we can even define functions inside a class
void changeName(String newName){
name = newName;
} // using this function we can modified pre exiting name
Student(){
this.rollno = 73;
this.name = "Rohit";
this.marks = 87.5f;
}
Student(int rollno, String name, float marks){
this.rollno = rollno; or roll = rollno;
```

this.name = name; or naam = name; // doesn't have to be the same variable name

this.marks = marks; or score = marks;

}

}

// If you are using same variable name for both in argument and in refference variable, you have to use this. (dot)

this → here 'this' keyword nothing but defining in the place of of Student1 and student2

So, student1 and student2 will be automatically putted to in the place of 'this' accordingly.

* when the new object is being created it's go inside the constructor *

For example:

object with no parameter passing to constructor as a argument, will look inside the constructor that has no argument passing to it.

```
Student student2 = new student();
Student (){
 this.rollno = 73;
 this.name = "Rohit";
 this.marks = 87.5f;
        }
```

- And object with parameter passes to a constructor as an argument will look inside the constructor that has argument.

```
Student student1 = new Student(73, "Rohit", 87.5f);
Student (int rollno; String name, float marks){
    // Some code
    // Some code
        }
```

- Other than that you can call the greeting function like this. ↓

```
student1.greeting()
```

You will get Output : Hello! My name is Rohit

```
// because incase of student1, we already
   have mentioned name.
   this.name = name; // where we had
∴ this.name = Rohit;   passed 'Rohit' as
                       argument in place
                       of name
```

- we can also use change name function like this. ↓

```
student1.changeName("Himesh");
```

after this if you use greeting function As, the name will change from 'Rohit' to → 'Himesh'
The out put you will get → Hello! My name is Himesh

- **Construction Overloading:** when you call a construction with values it will call construction which have values as argument in it. And when you call a construction with zero values it will look into the construction which have no argument passing in it.

ex: If we take reference from previous example

This will → Student student1 = new Student(73, "Rohit", 87.5f);

Call this → Student(int rollno, String name, float marks){

// Some code

// Some code

}

And this will → Student student2 = new Student();

check into

this → student(){

this.roll no = 73;

this.name = "Rohit";

this.marks = 87.5f;

}

- Multiple construction (Analyzing):
- So, Now we have learn how to build a construction and A construction can be build with the same name as the class
- Now we will Analyze how we are multiple construction can be made and depending on what arguments an object have. It will pick what construction to use.

Let's Understand it with an example:

ex: Create a Construcror which take value from other object.

// when we calling this Construcror

```
Student (Student other){
    this.rollno = other.rollno;
    this.name = other.name;
    this.marks = other.marks;
}
```

this → random
other → student1

this properties will have access to these values

// creating a new object

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

// calling a Construcror to be the part of the object itself

```
class Student{
    int roll no;
    String name;
    float marks;
    Student(){
        this.rollno = 73;
        this.name = "Rohit";
        this.marks = 87.5f;
```

Now, if this replace by → random &
other replace by → student1

That means student1 will have accegs to values like,

student1.rollno = 73;

```
student1.name = Rohit;
student1.marks = 87.5f;
```

// If, you print it like ...

```
sout(student1.roll no);
sout(student1.name);
sout(student1.marks);
```

// You will get output like

Output :- 73
Rohit
87.5

* This is how we can create multiple constructor. and that's how multiple constructor work *

Moreover to that

- if you call the object with one argument, it will have constructor for that.
- if you call the object with three argument, it will go for the constructor that will have three argument

- Also,

```
student(int roll no, String name, float marks){
    this.roll no = roll no;
    this.name = name;
    this.marks = marks;
                }
```

using the above constructor template and passing the value in it, we can create as many object as we want.

// Student Manash = new Student(93, "Manash", 92.7f);

Internally it will be like,

Manash.rollno = rollno that we have passed as argument

Manash.name = name that we have passed as argument

Manash.marks = marks that we have passes as argument

> Note: Constructor don't have a return type because by default the type of the class itself is return type. So, no void you have to write.

'this' keyword is nothing but referring to the reference variable itself

// because we can't put the reference variable in the class, As class is the template, So for that Java defines 'this' keyword. It was the reference to the object on which the method was invoked. It('this') will replaced with the current object when you call it.

- Calling a Constructor from another Constructor

```
Student random = new Student(); //an object
Student(){ // this is an empty constructor
  this(19, "default Person", 97.3f); // this is how
                                     // you call a constructor
                                     // from another constructor
}

Student(int rollno, String name, float marks){
    this.rollno = rollno;
```

```
this.name = name;
this.marks = marks; }
```

// Now when you print it like

```
Sout (random.name); // You get
```

Output: default person

- why we don't use 'new' keyword for creating primitive data types ?

→ Because in Java Primitive data types like: int, float, char etc. are not implemented as an objects. As these are not object hence these are not stone in the 'Heap' memory, these are stone in the 'stack' memory only.

- How to allocate memory using 'new' keyword

→

// Suppose using 'new' keyword we have created an object like this ↓

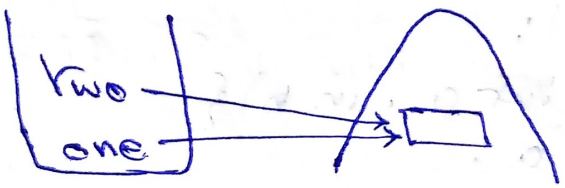
```
Student one = new Student();
```

// And we are assigning one to two like this ↓

```
Student two = one; // now both one & two
                   // both pointing to the same object
```

because:

first, we have created object one and it's pointing to an object in a heap memory. then, as you created two and assigned one to it, Now, two will point same object as one.



- n.v two doesn't copy anything of one it's just pointing to the same n.v as one

* Any change made by n.v one, will also lead to changes via two *

So, Now if you print it like:

~~sout~~ one.name = "Raghav";
sout (two.name);

Output: Raghav // You get out as modified by one

■ Wrapper class: Just like other classes wrapper class is also group of properties and functions.

// Now, if you do something like

Integer new = 45; // here, new is an object of type integer.

new. // using new dot we can use so many properties and functions, As new is now a wrapper class

● Now, Again

Ex:

```
int a = 10;
int b = 20;
Swap (a, b);
sout (a + " " + b);
static void swap (int a, in b){
    int temp = a;
```

```
    a = b;
    b = temp; }
```

// this will not swap the values of 'a' and 'b' and you will get Output: 10 20

because in Java there is no such thing call on in Java pass by reference variable doesn't work only pass by values work.

- 'final' keyword:

eg:

```
Integer a = 10; // now it's an object
Integer b = 20;
Swap (a, b); // now these are not pass by reference variable but by reference value
sout (a + " " + b);

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

This will still doesn't work, and you will get Output: 10 20 // It will still not get swapped

* because here, Integer is class & it's a final class. A final class value doesn't get modified once a value assign to it *

eg: Suppose everyones pay is going to be increase by 2%.

final int INCREASE = 2;

→ By convention, if there is a 'final' keyword, make sure n.v all in caps

* Using 'final' keyword you can prevent your content to be modified *

- Always initialize 'final' keyword while declaring
- Using 'final' keyword you can not make change in the value when it's premitive data type. But if it's not premitive data type you can make change in the value, but you can not reassign it.

ex:

final int INCREASE = 2; // You can make change here as it's premitive data type

final Student Kunal = new Student();

Kunal.name = 'new name'; // you can change value

Kunal = other object // but you can not reassign another object

- Garbage collection:

~~In java~~ garbage collection is the process of reclaiming the runtime unused memory by destroying unused objects.

As, in case of C we use free() & in case of C++ we used delete() functions

But, Java in being performed automatically, So Java Provide better memory management

So, Java Garbage Collection is the process by which Java programs perform automatic memory management.

the disadvantage of languages like; C & C++ is that if forgetting to destroy useless objects will eventually leads to what we call 'memory leaks'.

after a certain point, memory won't be available anymore to create new objects, and the entire application will terminate due to Out-of-Memory-Errors.

In Java garbage collection happens automatically during the lifetime of a program, eliminating the need to de-allocate memory and therefore avoiding memory leaks.

when Java programs run on the JVM, objects are created in the heap space, which is a portion of memory dedicated to the program.

Heap Memory → consist of two type of objects. ——

① Alive
② Dead

① Alive: Alive objects are used and referenced from some where else in your application.

② Dead: Dead objects are no longer used or referenced from anywhere in the application. These objects are detected by the GC and deleted to free up memory.

* Garbage Collector detect and delete the unused objects from heap space *

▶ How GC Operates:

- Three main phases to GC: Now, to perform garbage collection or write your garbage collector you should include in your implementation three main phases...

① Marking objects as Alive: In these step the GC identifies all the alive objects in Memory by traversing the object graph. when GC visit an object it marks it as accessible hence alive. All the objects that are not accessible consider candidate for garbage collection

2nd phase – Sweeping Dead objects: After marking all objects as alive or dead. the sweeping phase release the memory fragments which contain these dead

objects.

Final or 3rd Phase – Compact Remaining objects:

In these phase the remaining objects in memory are compacted. this phase make sure the memory is compact after garbage collector deletes the dead objects. So, the remaining objects are in contiguous order at start of the heap.

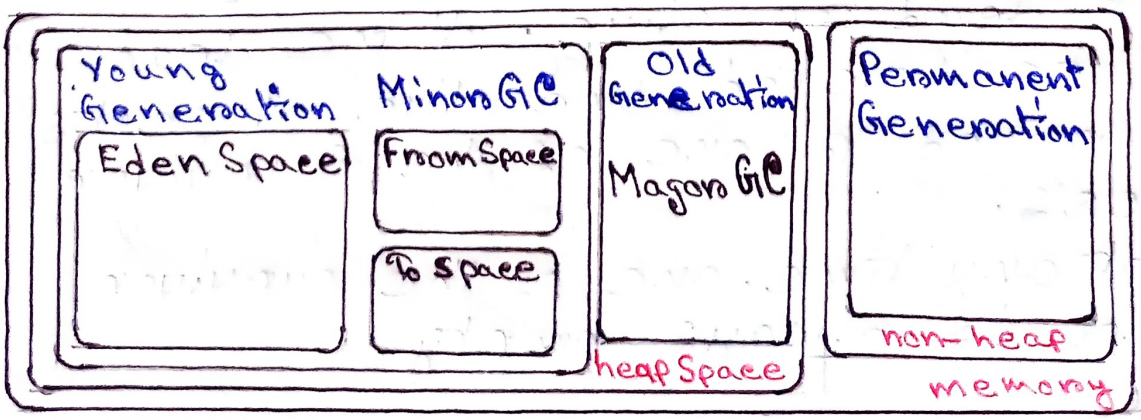
But, Having to mark and compact all the objects in a JVM every single time is inefficient,

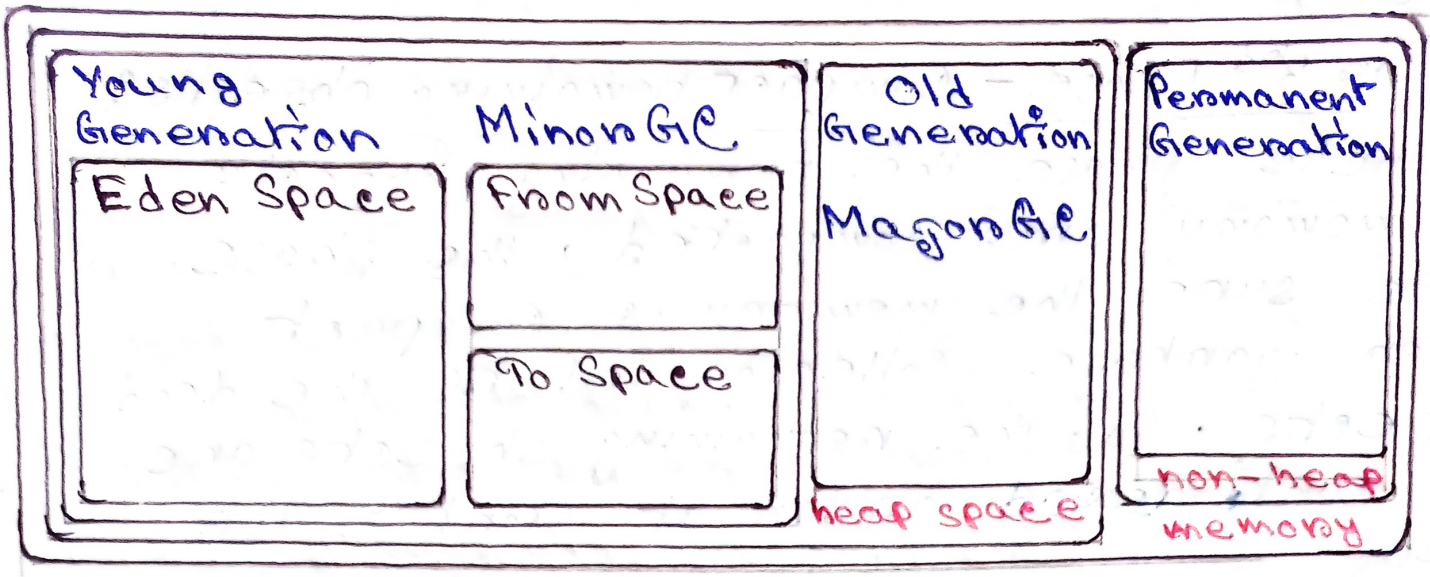
And to solve this Java garbage collector Implemented

- Generational Garbage Collection:

As more and more objects are allocated, the list of objects grows, leading to longer garbage collection times.

~~se~~ most objects have a very short life span. to benefit from these findings the heap memory area in the JVM was divided into three sections.





1. Young Generation: the young Generation which consists of newly created objects and is further sub divided into Eden space and the Survivor spaces, ~~edit~~
   - All the new objects that start in the Eden and initial memory is allocated to them.
   - If an object survives one GC cycle then they are moved to Survivor Space
   - when objects are garbage collected from the Young Generation it is a Minor Garbage Collection event
   - A Minor GC is performed when the Eden space is filled with objects either dead or alive, all the dead objects deleted and alive objects are moved to one of the Survivor space's.
     So, at any time, one of the survivor spaces is always empty

- when the surviving objects reach a certain threshold of moving around the survivor spaces, they are moved to the old Generation.

(2) Old Generation: Old Generation is one of the three section in the heap memory.

- Objects that are alive long enough is moved ~~to~~ from young generation to old generation. It contains objects that have remained in the survivor spaces for a long time.
- When the objects are garbage collected from old generation, it's a Major Garbage collection event.

(3) Permanent Generation: Finally we have Permanent Generation, Metadata of classes and methods are stored in the Permanent Generation.

- It is populated by the JVM at runtime based on classes in use by the application.
- Since Java 8, the MetaSpace memory space replaced the PermGen space, the implementation differs from the PermGen as the space of the heap is now automatically resized.
- This avoids the problem of applications running out of memory due to the limited size of the PermGen space of the heap.

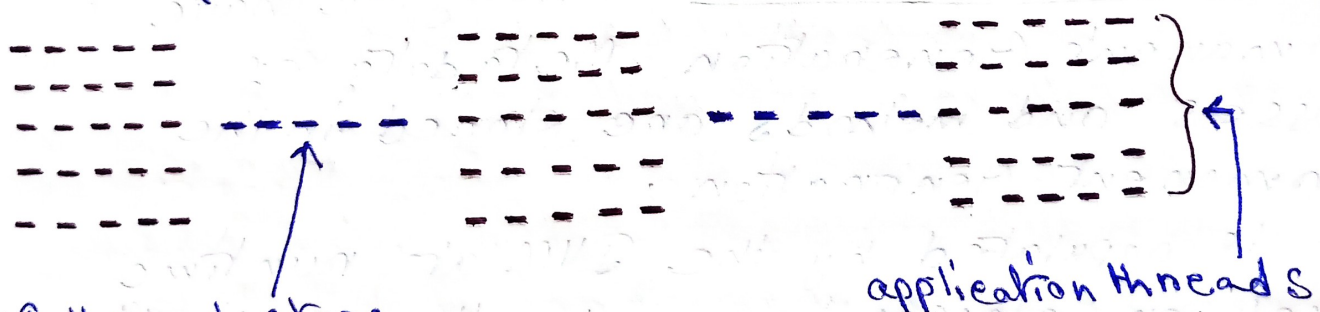
- The Meta space memory can be garbage collected and the classes that are no longer used can be automatically cleaned when the metaspace reaches its maximum size.

Now, that we know how GC operates, Let's take a look at types of Garbage Collectors available at JVM

▶ Types of Garbage Collectors:

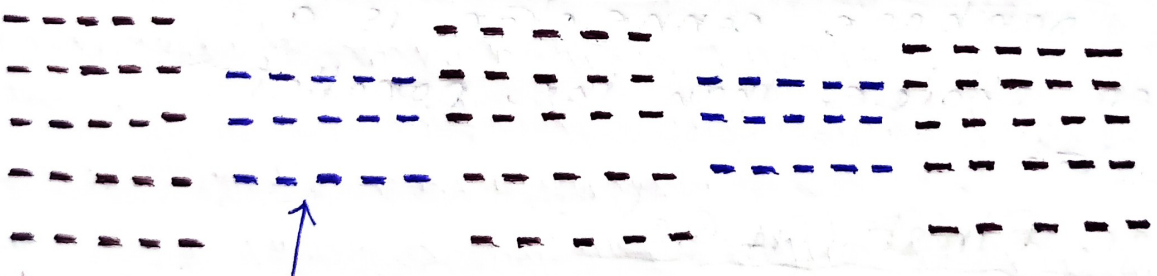
① Serial GC: This is the simplest implementation of GC and is designed for small applications running on single threaded environments all garbage collection events are conducted serially in one thread

- when it runs it leads to a stop-the-world event where the entire application is paused



② Parallel GC: The pair parallel GC. The parallel collector is intended for applications with medium-sized to large-sized data sets that run on multi-threaded hardware, multiple threads are used for minor garbage collection in the Young Generation and the single thread is used for major garbage collection in the old generation.

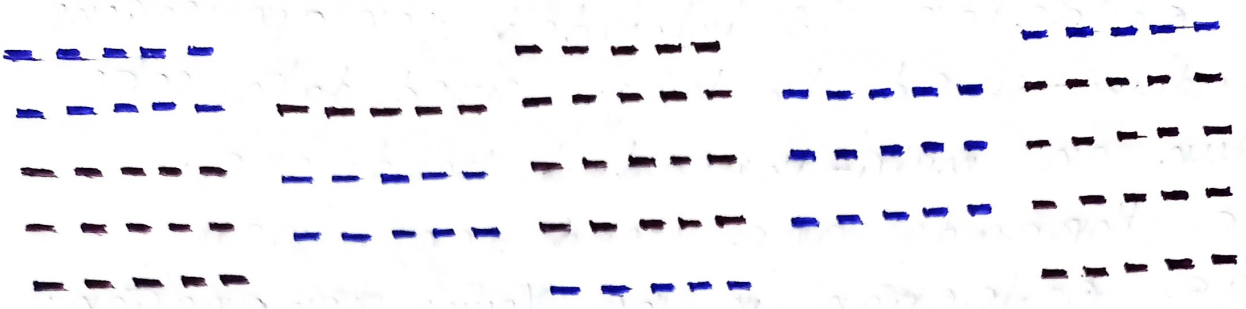
- Running the parallel GC also causes a 'stop the world event and freezes the application, so since it is more suitable in a multi-threaded environment it can be used when a lot of work need to be done and long pauses are acceptable.



multiple threads are used for minor garbage collection in the Young Generation and a single thread is used for major garbage collection in the Old Generation.

also causes a "stop the world event" and the application freezes.

③ Concurrent Mark Sweep (CMS GC) :

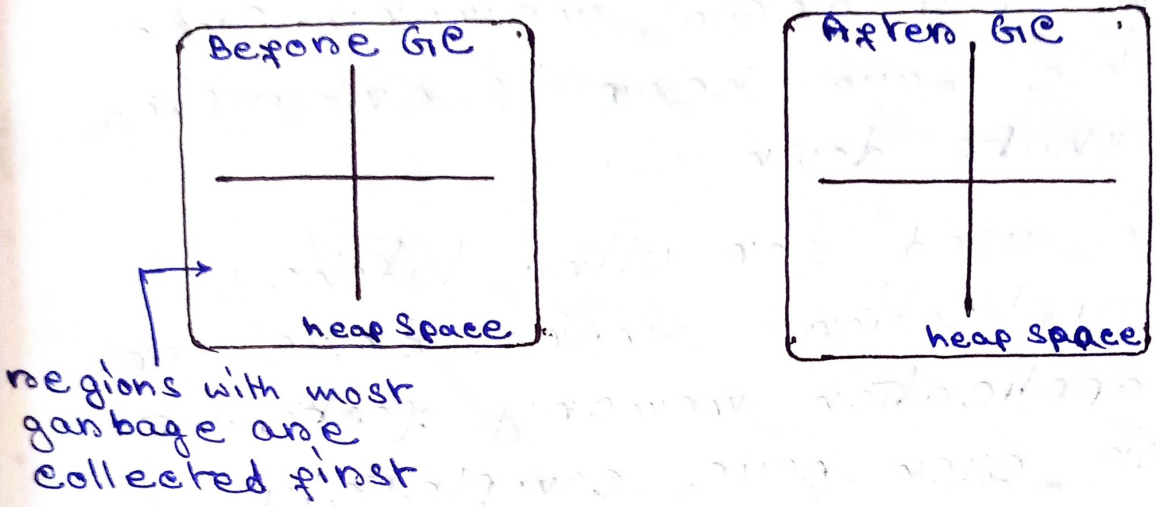


- Multiple threads are used for minor and major garbage collection.
- CMS runs concurrently alongside application processes to minimize stop-the-world events, because of this the CMS collector use more CPU than other GCs.
- So, If you can allocate more CPU for better performance then the CMS garbage connector is a better choice than the parallel collector

④ Garbage First (G1 GC) :

- Also known as default collector by Java
- G1 GC was designed for multi-threaded applications that have a large heap size available, under the hood it works quite differently compared to older garbage collectors

- G1 is generational, but it does not have separate regions for young and old generations, instead, each generation is a set of regions, which allows resizing in a flexible way.



- It partitions the Heap into a set of equal size regions and uses multiple threads to scan them.
  A region might be either an old region or a young region at any time during the program run
- After the marked phase is completed, G1 knows which regions contains the most garbage objects. If the user is interested in minimal pause times G1 can choose to evacuate only a few regions, if not G1 might choose to include more regions since G1 GC identifies the regions with the most garbage and performs garbage collection on that region first, it is called garbage first

(5) Epsilon GC :

- Epsilon do nothing garbage collecton that was released as part of JDK 11.
- It handles memory allocation but does not reclaim any actual memory. Once the available Java heap is exhausted, the JVM shuts down
- It can be used for ultra latency sensitive applications where developers know the application memory footprint exactly or even have completely garbage-free applications.
- The use of the Epsilon GC and any other scenario. is otherwise discouraged

(6) Shenandoah GC :

- Shenandoah is a new GC that was released as part of JDK 12
- Shenandoah's key advantage over G1 is that it does more of it's garbage collection cycle work concurrently with the application threads
- G1 can evacuate its heap regions only when the application is paused. while Shenandoah can relocate objects concurrently with the application.

- Shenandoah can compact live objects clean garbage and release RAM back to the OS almost immediately after detaching free memory. Since all of this happens concurrently while the application is running Shenandoah is more CPU intensive

⑦ ZGC :

- ZGC is another GC that was released as part of JDK 11 and has been improved in JDK 11
- ZGC is intended for applications which require low latency or use a very large heap.
- The primary goals of ZGC are low latency, scalability and ease of use. To achieve this, ZGC allows a Java application to continue running while it performs all garbage collection operations thus ZGC brings a significant improvement over other traditional GCs by providing extremely low pause times.