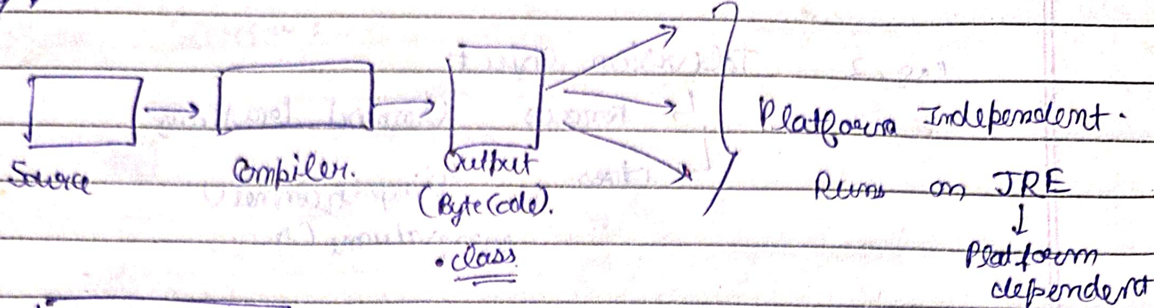
Page No.
Date

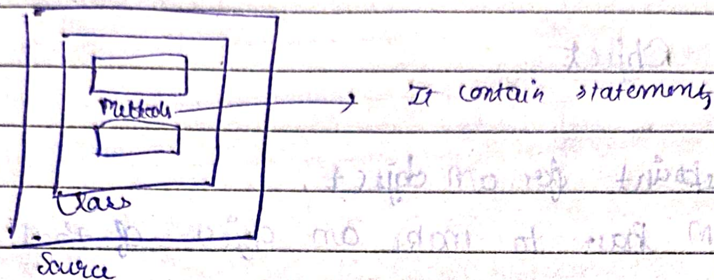
- 00, 03, -26

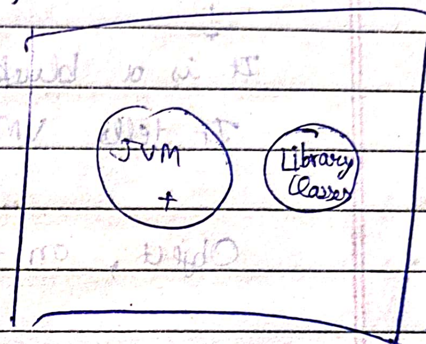
254, 72, 95

= 125, 65, 847

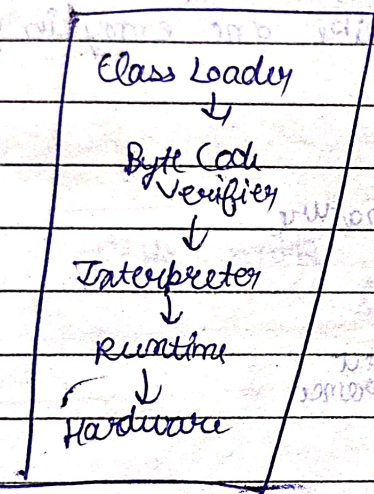
JAVA







JRE



Runtime

JDK = JRE + Development Tool

⇒ Object-Oriented Design (Objectville)

When designing a class,

- Think about the objects that will be created from that class type. Think about:
  - → Things object knows } instance variables
  - → Things object does } methods

e.g. Shopping cart
- ↳ knows content
- ↳ does addtoCart()
  RemoveCart()

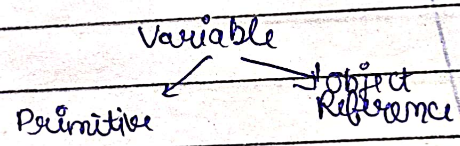
Date

e.g. 2 Television Object
↳ knows channel locations
↳ does changeChannel()
changeVolume()

- Class vs Object
↓
It is a blueprint for an object.
It tells VM how to make an object of that particular type.

Object, on the other hand, is like one entry in your address book

- Variables → know about their nature



Used as Instance variables
Local Variables
Arguments
Return Types

→ Primitives
↳ byte 8 bit
short 16 bit
int 32
long 64

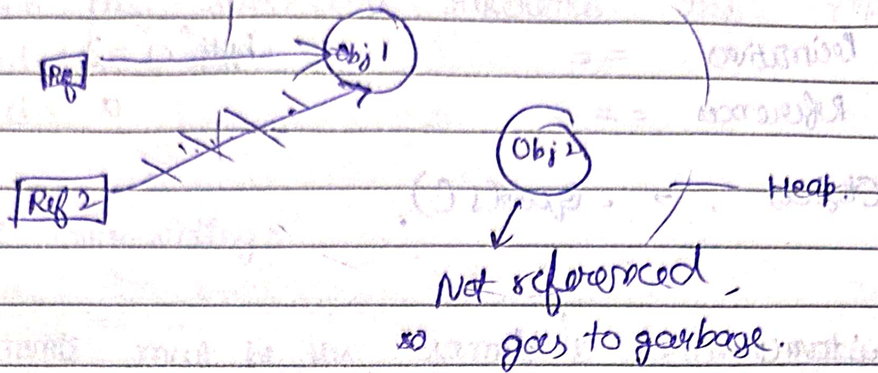
32 → float
64 → double

char
boolean

→ Object Reference
There is nothing like object variable, only references.

Date

→ Garbage Collector :



Ref 2 = null.

But, obj1 is not null.

→ Encapsulation :

Always mark getters & setters as ~~private~~ public

& instance variables as private

☆ Instance variables get a default value

integers 0
float 0.0
boolean F
Reference Null

i) Local variables are declared within a method
Instance a class

ii) Local variables must be initialized before use

iii) Method Parameters are also local.

Date

- Comparing Variables

Primitives ==   byte a=3   int b=3
                                 a==b // true

References ==

Objects , → .equals().

- Inheritance

→ Do use inheritance when one class is a more specific type of a superclass.

→ Do use it, when you have behaviour that should be shared among multiple classes of the same type.

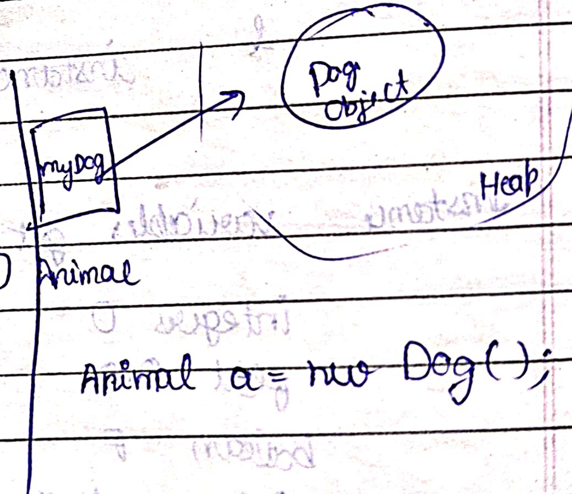
→ Do NOT use it just to reuse code from another class. IS-A test must be passed.

- Polymorphism

→ Can make polymorphic arrays

Animals[] anim = new Animals[5]

anim[0] = new Dog();
anim[1] = new Wolf();



→ Can have polymorphic arguments & return types

```
class Vet {
    public void giveShot (Animal a) {
        a.makeNoise();
    }
}
```

```
PSVM {
    Vet v = new Vet();
    Dog d = new Dog();
    Hippo h = new Hippo();

    v.giveShot(d);
    v.giveShot(h);
}
```

* Just remember,
we can pass superclass references (inside) the reference of subclass object.

- Method over-riding:

1. Arguments must be the same, & return types must be compatible.

2. The method can't be less accessible.

- Method Over-loading:

1. Arguments change or their data type.

2. Only return type change can't do overloading.
3. Access levels can be varied.

- Serious Polymorphism

1. One requirement in certain design patterns is not to create parent class objects. This behaviour can be achieved using:
abstract classes.
~~prevent " new <class name> "~~ to ever occur.

Animal a = new Dog() ✓
Animal a = new Animal() X

is not abstract.

→ Concrete class on other hand

→ Abstract class has no use, no value only can be extended or have static members (later)

2. Methods can be abstract too, only possible inside abstract class.

→ It has no body.

public abstract void eat ();

→ It might not look that it has some purpose, but it allows for a set of protocols.

→ They need to be implemented by first concrete class (mandatory), or maybe by an abstract class (not mandatory)

- Class Object:
  It is the superclass of everything.

→ Every class that doesn't extend it explicity, extend it indirectly.

→ So, what's in the object class?

equals (object o)
hashCode ()
get Class ()
toString ()

~~and~~ & many more

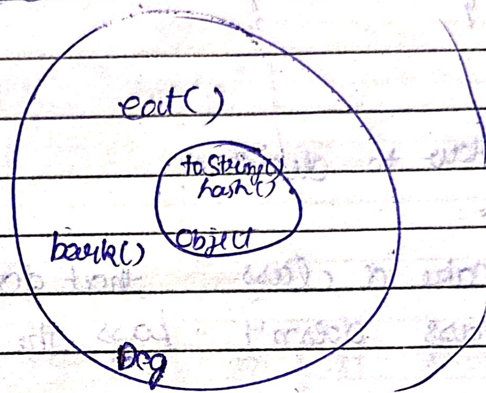
→ It can be used to :

- i) Act as polymorphic type for methods
- ii) Provide real methods code that all objects in Java need at runtime.
- iii) Thread related (later)

Notes Object o = new Dog()

o.bark() ✗

b/c there is confusion since object is superclass of everything.

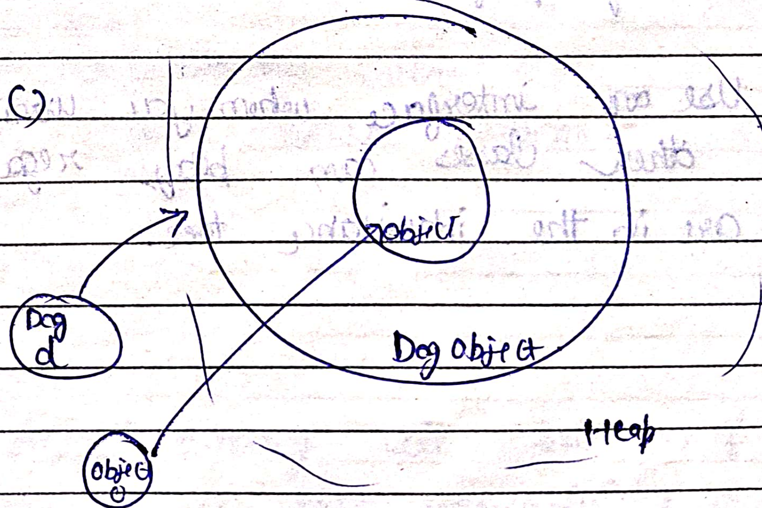


Notes Dog d = getObject(); ✗

again not possible

Dog d = new Dog()

Object o = d.



→ Casting can be done for sure

Dog d = (Dog) getObject();

"instanceOf" operator is helpful.

Page No.
Date

- Interface: (100% pure abstract class)

```
public interface Pet {
    public abstract void beFriendly();
    (public: not required (implicit))
}
```

How to decide?

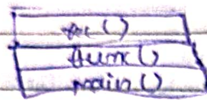
1. make a class that doesn't extend anything when your new class doesn't pass the IS-A test for any other type.

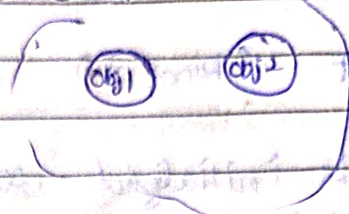
2. Make a subclass only when you need to make a more specific version of a class & need to override or add new behaviours.

3. Use an abstract class when you want to define a template for a group of subclasses.

4. Use an interface when you want to define a role that other classes can play, regardless of where those classes are in the inheritance tree.

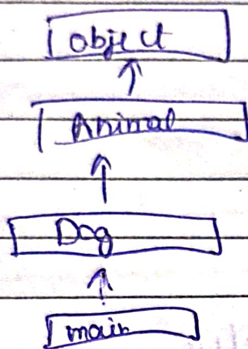
⇒ Garbage Collection: (& constructors)





Stack

Heap (Also called Garbage Collectible Heap)



(Constructor Chaining)

- Either write super(args) explicitly on first line of constructor or do nothing.

- this() is used to call other constructors. Note: super() & this () can't exist together in a constructor.
  this() must be the first line.

- Life : The entirety of an variable living
  scope: Currently on top of stack frame.

⇒ static & final

- static variables → shared among all instances of the class
- static methods makes use of static variables only, can be called without creating an object.
  Also, they can't access non-static method.

- static final
  ↓
  global constant (kind of).

  needs to be initialized at the time of declaration
  or
  static initializer must be used

```
static {
    CONST = 3;
}
```

- Final variable can't change it's value
  final method can't be overriden
  Final class can't be extended

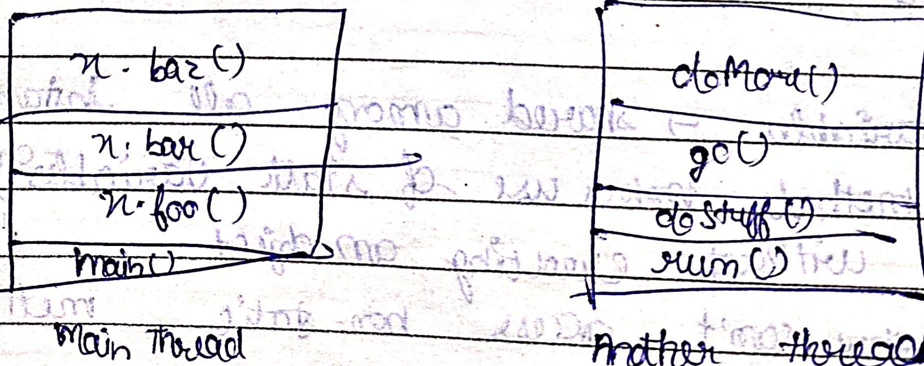
- Private constructor means no object can be created

⇒ MULTI-THREADING

```
Thread t = new Thread();
t.start();
```

} It has no job ☹

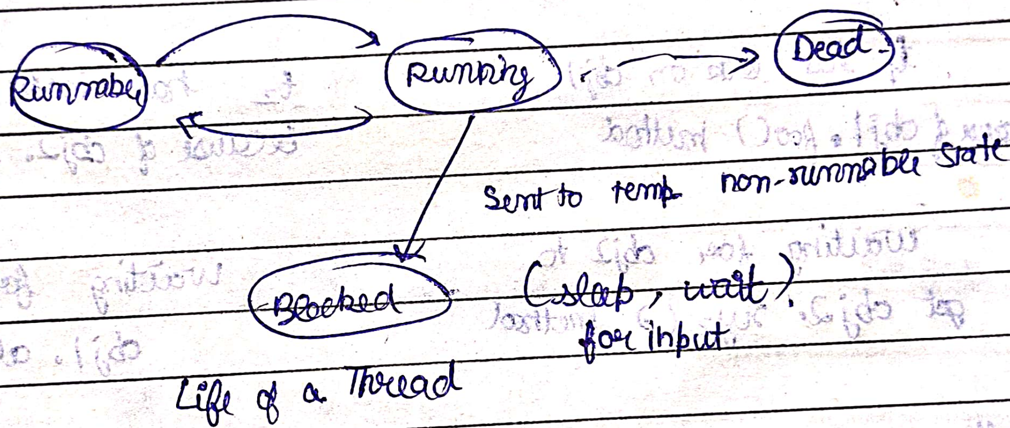
→ each thread has a separate call stack.



→ Launching a Thread with a Job.

```
Runnable threadJob = new MyRunnable();
Thread t = new Thread(threadJob);
t.start()

Class MyRunnable implements threadJob {
    public void run() {
    }
}
```



Life of a Thread

Thread Scheduler decides what to do, but there is a sleep() method.

- Thread.sleep(miliseconds) (Must be wrapped in try/catch)

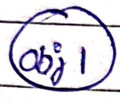
- t.setName(String), Thread.currentThread().getName();

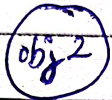
- "synchronized" keyword helps in creating locks on an object method, but lock is obtained on object

If there are two synchronized methods, a key is needed by thread to enter any of them.

Date

- every object has a single lock, with a single key for that lock.
- Even if object has more than one synchronized method, there is still one key.
- Deadlock





t_1 has lock on obj1 because of obj1.foo() method

waiting for obj2 to get obj2.xyz() method

t_2 has lock on obj2 because of obj2.bar() method

waiting for obj1 to get obj1.alpha() method.

all foo, bar, xyz, alpha are synchronized.

Note: There are two objects.

JAVA doesn't have deadlock preventing mechanism, so its all upto design by programmer.