Java:

Arrays: → fixed in size

```
class Array
{
public static void main(String[] args)
{
int a[] = new int[100];        double values[] = new double[50];
String words[] = new String[]{"my", "Name", "is"};
System.out.println(words[2]);
}
}
```

Both are same.

```
String words[] = new String[3];
words[0] = "my";
words[1] = "Name";
words[2] = "is";
```

Control flow:

```
class control
{
public static void main(String[] args)
{
boolean hungry = true;
if(hungry)
{ S.o.p("I am starving"); }
else{
S.o.p("not hungry");
}
```

Switch ( ) {
↓ Condition

```
Case 1:
        break;

Case 2:
        break;
  .
  .
default:
        break;

}
```

---

methods:

System.out.println ();
→ method.

System → Class
out → Variable → datatype of out is PrintStream.

public final static PrintStream
out = null;

clear
Developer
manager
Tester

→ method signature

public static void main (String[] args)

public → Access modifier → used to manage visibility of method

Class and objects

```
public class Human {
    String name;
    int age;
    int height;
    String eyecolour;

    public void speak()
    {
        Sy.o.p("Hello");
    }

    public void eat()
    { s.o.p("eating");
    }
    public void work(){
    S.o.p("Working")
    }
```

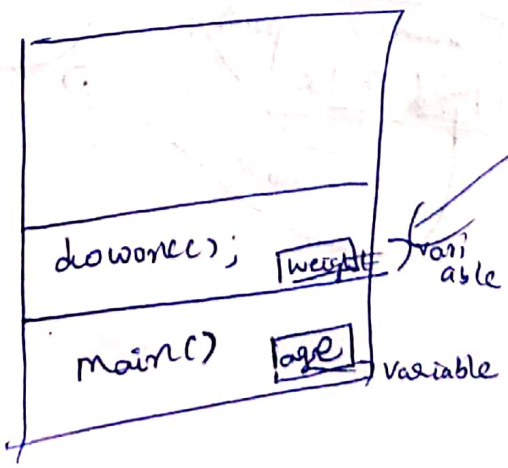
```
Human h = new Human();

public class Earth {
    psvm(String args[]) {
        Hum h = new Human();
        h.name = "Tom";
        h.age = 25
        h.height = 72;

        h.speak();

        Human j = new Human();
        j.name = "Joe";
        j.age = 27;
        j.height = 66;
        j.speak();
    }
```

Stack:



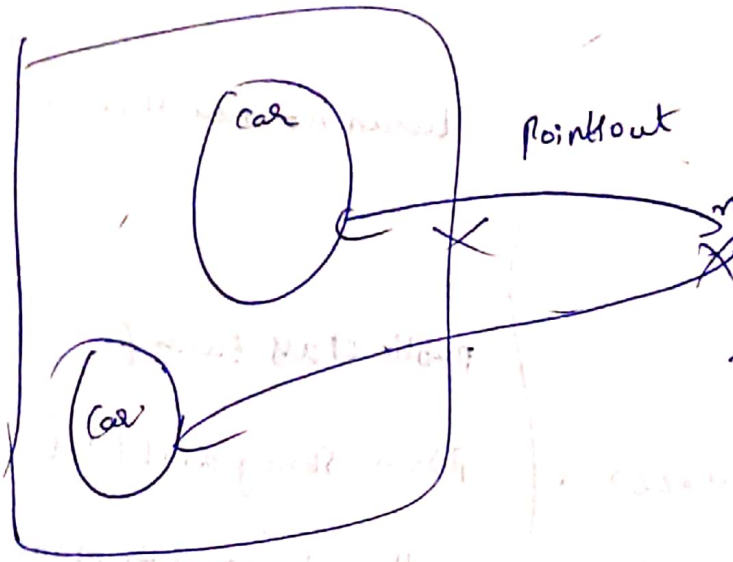

```
doWork()
{
float weight = no.3;
doMore();
```

once method doWork is executed it will be pop() after execution main() it will cleared

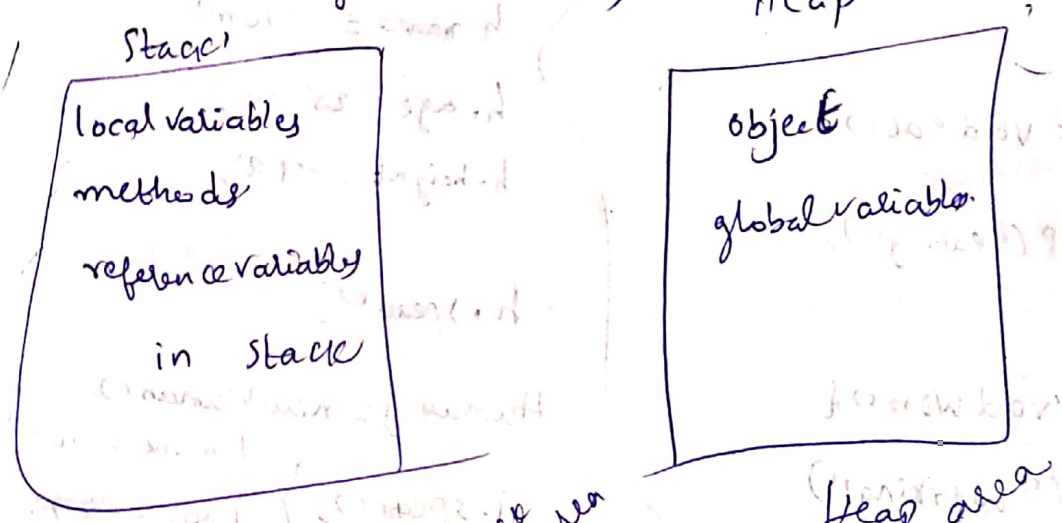
Heap memory

Car mycar;

mycar = new Car();



mycar = new Car();



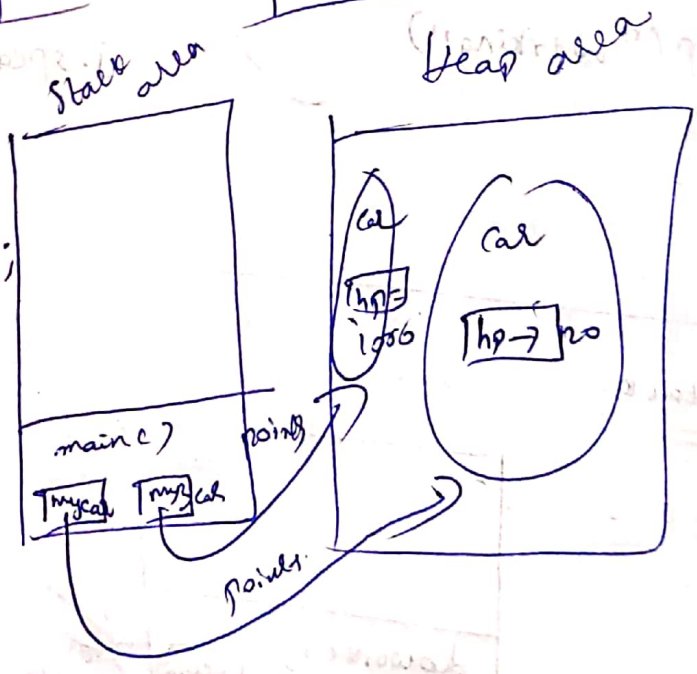
int hp;

Car myCar = new Car();

my.car.hp = 120.

Car myBcar = new Car();

myBcar.hp = 1000



Engine, myEngine;

Engine.bigEngine = new Engine();

mycal.myEngine = bigEngine

↓

address of Engine assined to myEngine.

Inheritance:

```
class Animal
{
  int age;
  String gender;
  int weight;
  Animal(int age, String gender,
         int weight)
  {
    this.age = age;
    this.gender = gender
  } this.weight = weight;
```

```
public class Bird extends Animal {
  public Bird(int age, String gender) int
                                      weight
  {
    super(age, gender, weight);
  }
  public void fly()
  {
    S.o.p("fly");
  }
}
```

```
class Zoo
{
  public static void main (String)args[])
  {
```

Interfaces:

Any class implements interface.

```
Public interface flyable {

  // Abstract methods.
   Public void fly();

}

public class sparrow extends Bird implements flyable {

public sparrow (int age, String gender, int weight) {

super(age, gender, weight);

}

@override

  public void fly()

  {

  S.o.p("sparrow flying high");

  }
```

Interface is a contract any class implements interface

We can only extends one class.

We can have multiple interfaces.

Abstract class:

1. we can only extend.
2. It has abstract method.
3. Use abstract keyword.
4. You cannot create instance of abstract class.

```
public abstract class Animal
{
  Public void move();
}

Public class Bird extends Animal
{
  public void move()
{ S.o.p("move");
}
```

```
class zoo
{
Public static void main(String[]
                        args)
Animal Sparrow = new Sparrow
                    (1, "m", 4);
Sparrow.move();
```

String class:-

Class String {

P s v m (String[] args)

{

String a = "hello";

System.out.println (a);

}

}

S.o.p (a.length());

String name = "R | O | B | E | R | T"

0 1 2 3 4 5

Str =

String str = "ABCDEFGH";

Syo (str.substring(1)); from index number 1 to remaining

Str.substring (0, 2) → "AB"

Str.substring (1, 3) → "BC"

Str.substring (3, 6) → "DEF"

Str.substring (3, 7) → "DEFG".

String a = "hello";

String b = "there";

if (a.equals ("hello")) { } → To compare values

```
if(b.equals("there"))
  {s.o.p(" print there");
  }
```

```
if(b.equalsIgnoreCase("There"))
  { s.o.p("print there");
  }
```

```
system.out.println(a.charAt(4));
```

```
char mychar = 'H';
char mychar2 = 'j';
char mychar3 = 'j';
```

not negative value.

```
int b = str.indexOf("thre");
int c = str.indexof("there", 0);
int d = str.indexOf("there", 7)
```

b) second index value of there

whileloops:-

```
int c=0
while(cc=100)
{s.o.p("Hello")
c = c+1;
}
```

```
class Demo {

psvm (String args[]) {
{
String str = "we have a
large inventory falling in" +
" the category : apparel and the
slightly" + " more in demand
Category : makeup along with
the category : furniture

print Categories (str);

}

Public static void printCategories (String str) {

int i=0;
while(true) {
int found = string.indexof("category:", i);
if (found == -1) break;
int start = found + 9;
int end = string.indexOf(" ", start);
s.o.p (" string.substring(start, end));
        i = end + 1;
```

Use:

substring (string start, string end)

indexOf (string s, int i)

while (true) {

for loop:

```
for(int i=0; i<100; i++) {
    s.o.p("i = "+i);

}
```

```
class loop {

psvm (string[] args)

{ string name = "sfdfewencsoidfhsjtbf";

  for(int i=0; i<name.length(); i++)

  { s.o.p("char :"+ name.charAt(i));

  }
}
```

```
for(int i= name.length()-1; i>=0; i--)

  {
    s.o.p("char i" + name.charAt(i));

  }

for(int i=0; i<=100; i++)

{ if(i%2==0)

  { s.o.p(i)

  }

}
```

```
class x
{
int x=500;
void xyz()
{
System.out.println("xyz");
}
Static int a=100;
Static void abc()
{ S.op("abc");
}
class Demo5
{
psvm (String args[])
{ x.abc();
S.op(x.a);
X x=new X();
x.abc();
S.op(x.s);
}
}
```

```
class x
{
x()
{
S.op("x is a constructor");
}
void abc() {
S.op("abc is a method");
}
}
class Demo5
{
psvm (String args[])
{
X x=new X();
x.abc();
```

constructor

1. class and constructor name must be same.
2. No return type
3. every item it will create memory
4. once per obj
5. We can't write using static.

method

1. class name and method name must be same or other
2. return type.
3. many items per obj.
4. We can write non access modifier

```
interface Demo
{
    void show();
    void display();
}

abstract class X
{
    abstract void loan();
    Public void new()
    {
        S.o.p("method");
    }
}

class Y extends X implements Demo
{
    public void loan()
    {
    }
    public void show()
    {
    }
    public void display()
    {
    }
}
```

```
class Demo {
public static void main(String [] args)
{
Y y = new y();
y.loan();
y.show();
y.display();
}
}
```

class.	Abstract class	Interface.
Only normal method.	Both. Abstract & normal method.	Abstract methods
object created	no object	no object.
no multiple inheritance.	no multiple inherit	multiple inheritance
has Constructor.	has constructor	no constructor
int a.	int a;	

Jar folder:

Java -jar no filename.

Jar -tvf .mario.jar
↓ ↓ ↓ ↓
table of contents → file → verbose → filename.

+vf → cf
x → extract

Main-class: Exampleprogram. → Saves as manifest.txt

jar. -cvfm → manifest myprogram.jar manifest.txt

manifest.mf.
↓
which class Contains main method.

create → file → verbose

..

jar -cvfm myprogram.jar manifest.txt *.class

java -jar myprogram.jar

Jar - Java Archieve

↓ collection of Java files.

jar -cvfm

jar -cvf hell.jar a.class. b.class. c.class

Files :

```
File file= new File("myfile.txt");
Scanner input = new Scanner(file);
while (input.hasNextLine()) {
    String line =input.nextline();
    s.o.p(line)
}
input.close();
```

```
Public class Runtime extends Exception {
    Public Runtime(String message) {
        Super(message)
    }
}
```

File processing with Buffered reader & finally block:

```
class Application {

File f=new File("file.txt");

try
{
FileReader fr= new FileReader(f);
BufferedReader br = new BufferedReader(fr);
String line = br.readLine();

while (line!=null) {

S.o.p(line);

line = br.readLine();

    }

} catch (FileNotFoundException e) {

Sop("Filenot found");

} catch (IOException e) {

S.o.p(o.file.getName());

}
   finally {
     try{
     buf.close();

     } catch(IOException e) {
          S.o.p(file.getName());
```

catch (NullPointerException ex)

{ S.o.p ("file never opened" + ex);

}

try with resources:

Pass filereader and Buffered reader

try (FileReader fr = new FileReader (file);

BufferedReader br = new BufferedReader (fr));

→ It will automatically close filereader and BufferedReader

no need of fr.close();

br.close();

Autocloseable &

Public interface Autocloseable {

}

Public interface closeable extends Autocloseable {

}

→ closes the streams and releases any System resources

```
class App2 implements Autocloseable{

@Override

public void close() throws Exception{

S.o.p("closing");

}
}

Public class Demo

{
psvm(String[] args)

{
try(App2 a = new App2();){

}
catch(Exception e) {

e.printStackTrace();

}
}}
```

Collections:

ArrayList is a dynamic or resizable array we can increase its size by 50%.

```
class App {
{
psvm(String[] args)
{
ArrayList words = new ArrayList();
  words.add("Hello");
  words.add("there");
  words.add(10);
  words.add(12.00);
  words.add('H');
  object item1 = words.get(0);
  object item2 = words.get(3);
  sop(item1);
}
}
```

String i1 = (String) words.get(0)

String i2 = (String) words.get(3)

<> → Generic for typesafety

→ collection of nodes linked together

```
LinkedList<Integer> n1 = new LinkedList<Integer>();
  numbers.add(10);
  numbers.add(20);
Sop(numbers);
```

```
for(int number : numbers) {
    s.o.p(number);
}
```

Linkedlist is faster for manipular but slower for retrieval.

Arraylist is faster for retrieval, slower for manipulation

Traversing Lists and custom types:

```
class App{
psvm (String[] args)
{

ArrayList<Vehicle> al = new
                ArrayList(Vehicle> ();
                                  By default 10
al.add(new Vehicle("Ford", "Acc",
                    10000);
```

```
class Vehicle
{ String make;
  String model;
  int price;
}
```

```
al.add(new Vehicle("Bmw", "Ferrari", 12000);
```

```
for( Vehicle v : al)
{
    Sop (v);
```

```
public static void
printElement (List<Some> l2)
{
for(int i=0; i<someList.size();
                    i++)
{ s.o.p (s.omelist.get(i)); }
```

Set: It is an interface prevents duplicates:-

```
class Application
{
psvm(String[] args)
{
HashSet<Animal> al = new HashSet<Animal>();
al.add(new Animal("dog", 12));
al.add(new Animal("cat", 10));
al.add(new Animal("dog", 12));
al.add(new Animal("Lion", 13));

for(Animal a : al)
{
  sop(a);
}
```

```
sop(animal.equals(animal));
sop(animal.hashcode());
sop(animal.hashcode());
}
```

) duplicates not allowed.

LinkedHashSet is used for getting elements in insertion order

```
class Animal{
String name;
int age;
}
```

Comparable Interface:-

Map:- Store data in form of key value pairs.

key can not be duplicate.

Use put method to add data

```
class App {

psvm (String[] args)

{

Hashmap<String, String> dictionary = new HashMap<String,
                                     String>();

 dictionary.put("Hello", "Aii");

 dictionary.put("222", "Peter");

 dictionary.put("33", "John");

 for (String word : dictionary.keySet()).

    {
     Sop(dictionary.get(word));

    }
 }
```

Entry set:

It returns a set view by mapping.

```
for ( map.entry <String, String> entry : dictionary.entrySet())

{
    System.out.println (entry.getKey());
    System.out.println (entry.getValue());
}
```

LinkedList follows insertion ordered like how we are adding elements we will get in same order

TreeMap or TreeSet in Sorting ascending order

If new value is inserted for old key,

old value is replace by new value

Generics:

```
class App {

psvm(String[] args) {

Container c = new Container(12, "Hello");

Container<Integer, String> container =

new Container<>(12, "Hello");

int val1 = container.getItem1();

String val2 = container.getItem2();

}

Public static Set<E> union(Set<E> set1,
                           Set<E> set2)

{

Set<E> result = new HashSet<E>(set1);

result.addAll(set2);

return result;

}

}
```

→ represents type.

1) To make class generic

```
class Container<i1, i2>

object item1;

object item2;

Public Container(i1 item1,
                 i2 item2)

{

}

Public void printItem()

{

Sop(item1);

Sop(item2);

}
```

1) // a generic class

Generics with wildcards:

```
class App {
psvm(String a[])
{
Object myobject = new object();
String myvar = "hello";
myobject = myvar

Employee emp = new Employee();
Accountant acc = new Accountant();
emp = acc; // valid.
```

```
class Employee
{
public void work()
{ }
}
```

```
class Accountant
extends Employee
{
public void work()
{
}
}
```

```
ArrayList<object> employees = new ArrayList<object>();

ArrayList<Accountant> acc = new ArrayList<Accountant>();

employees = acc; // not valid

        ↳ wildcard.
ArrayList<?> e2 = new ArrayList<>();

ArrayList<String> ac2 = new ArrayList<String>();

e2 = ac2 // possible.

ArrayList<? extends Employee> e3 = new ArrayList<>();

ArrayList<Accountant> a3 = new ArrayList<Accountant>();
```

ArrayList<? super Employee> employee4 = new ArrayList<>();

↳ We can assign only parent or Employee

↳ not sub classes

public static void makeEmployeeWork(List<Employee> emp)

{

for (Employee e1 : emp)

{ e1.work();

}

}

Multithreading:-

multiple lines of code running at same time.

Threads need to communicate with each other.

Thread is essentially a task.

Thread is flow of program.

class App {

psvm(String[] args)

{

Task t = new Task();

t.start();

Sop("Hello");

}}

```
class Task extends Thread
{
public void run()
{
for(int i=0; i<1000; i++)
{ sop(i);
  }
 }
}
```

if one instance is called again it will show Exception in main thread.

If one thread is executed we cant use it. we need to create another one.

```
class Task implements Runnable
{
public void run()
{
Thread.currentThread().setName(this.name);
for(int i=0; i<10; i++)
{
sop("number:" + i + "  " + Thread.currentThread().getName());
 }
}
```

```
Thread t1 =
new Thread(t);
```

Thread Synchronization.

```
class Sequence.
{
int value = 0;
    public int getNext() {
synchronized(this) {
        value = value + 1;
        return value;
    }
}
}
```

Instead of Synchronized block.

```
Public int synchronized b
    int getNext() {
    }
```

```
class Worker extends Thread
{
Sequence sequence = null
public Worker(Sequence sequence)
{
this.sequence = sequence;
}
public void run()
{
  for(int i=0; i<20; i++)
  {
    Sop(sequence.getNext());
  }
}
```

Atomicity means, to complete full line or not.

multithreads completes only half

Synchronized is a lock.

```
class App {
psvm(String[] args)
{
Sequence sequence = new Sequence();
Worker w1 = new Worker(sequence);
w1.start(); } }
```

Thread safety with collections:

class Product {
int id;
String name;

}

Join()

Joins with main thread.

put method is threadsafe in producer consumer problem.

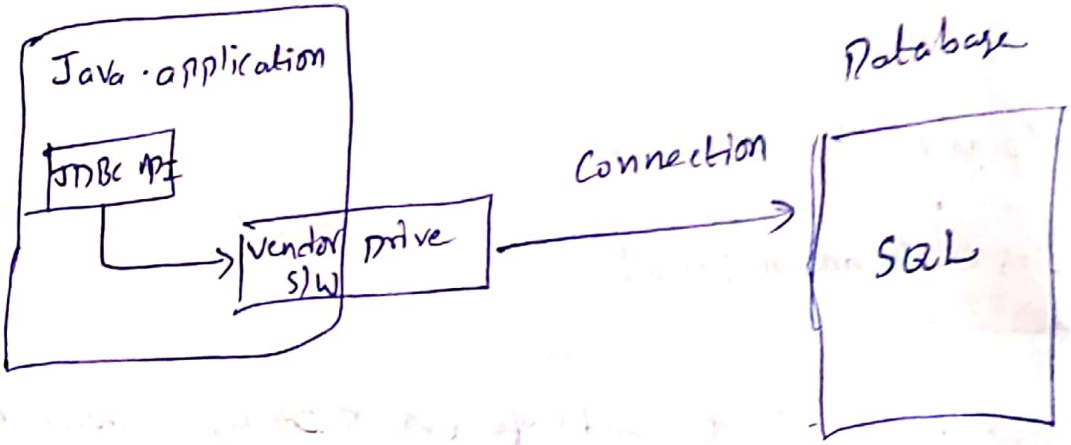
take() method remove or retrieve in BlockQueue.

JDBC:

Java DataBase connectivity:-

Connecting Java application with database:-

1. SQL, Sequential Query language




```
class JDBC {

psvm (String[] args) {

String url = "jdbc:mysql://127.0.0.1:3306/employee";
                                                    ↑
                                                database name

Connection con = DriverManager.getConnection(url, "root",
                                                "password");

Statement st = con.CreateStatement();

ResultSet rs = st.executeQuery("select * from employee");

  While(rs.next())

{  so.o.p(rs.getString("Salary"));

}

}
```

ResultSet rs = st.executeUpdate("insert in to emp values(800, 'Sales', 5500);");

↳ for inserting

Sop("Executed an inserted row");

for deletion:

st.executeUpdate("DELETE from emp where id=800");

for selecting we use st.executeQuery("sql query");

for insert & delete we use st.executeUpdate("sql query");

for updating st.executeUPDATE ("UPDATE Employees SET SALARY=6000 where id=600;");

Java Jshell:

jshell > System.out.println("Hello there ...")

no need of Semicolon.

jshell > String var = "hello".

var // output: hello.

jshell > /list.

1. System.out.println("Hello there")
2. String var = "hello".
3. var

jshell > void printTimes (String var) {

> for(int i=0; i<10; i++) {

System.out.println(var);

}

}

for block we need Semicolon

jshell > printTimes("hello");

jshell /edit printTimes. ⟶ we can edit the method or variable

jshell> /drop printTimes. → dropping the method.

jshell> /history

jshell> /exit

$ jshell, new instance of jshell.

jshell> /exit

$ clear

$ jsell -c /usr/Pathname/Desktop/Supermarijal

jshell> import com.printer.utils.*;

jshell> /type

jshell> /imports

jshell> Superprinter sp = new SuperPrinter();

jshell> sp.print()

Functional Interface:

@FunctionInterface.

```
class Robot {
public void walk()
{
Sop("Robo Walking");
}
}
```

```
class Human
public void walk()
{
Sop("Human Walking");
}
}
```

```
class App {
psvm(String[] args) {
Human hom = new Human();
Robot r = new Robot();

walker(new Walkable() {
public void walk()
{
Sop("Custom object Walking");
}
});
}
```

```
interface Walkable
{
public void walk();
}

public static void
walker(Walkable wek)
{
wek.walk();
}
```

.walker(() -> System.out.println("Custom object Walking"));

Functional Interface, interface with only one abstract method.

```
Walker ( () -> Sop("hello"));

Walkable block = () -> {
    Sop("custom object walking");
    Sop(" the object tripped");
    };
Walker(block);
}
```

Lambdas are block of code used to implement a method defined by functional interface.

```
interface Calculate
{
public void compute(int a, int b);
}
```

```
class APP
psvm {
Compute c = (a, b) ->
                a+b;

Sop(c.compute(4,6));
}
```

m();

```
(a,b) -> {
    if(a==0) {
        return 0;
    }
    return a/b;
};
}
```

Calculate cd

```
public String reverse()
```

```
rev = (s) -> {
    String result = "";
    for(int i = s.length(); i>=0; i--) {
        res.t = s.charAt(i);
    }
    return result;
};
}
```

```
interface StringWorker {
    public String work(String s);
}
```

```
public int factorial(int num) {

post d = (n) -> {
        int r=1;
        for(int i=1; i<=n; i++) {
            r= r*i;
        }
        return r;
    };
}
sop( d.compute(5));

interface mygeneric<T>{
    public <T> work(T);
}

mygeneric <String> any = (s) -> {
        string result = "";
        for(int i= s.length()-1; i>=0;i--) {
```

return result.

};

5. mygeneric("hello");

Built in functional Interface:

```
Public class Car {

  String make, model, color;

  int price;

Public void print(Car c)
{
S.o.p(this)

}

}
```

```
class APP {

psvm(String[] als)
{

List<Car> C = new ArrayList
                <>();

C.add(new Car(  ));

C.add(new Car(  ));

};
```

public static void printCar

predicate<T> predicate

1) functional interfaces

↓

Contains lambda expressions.

Function<Car, String> priceAndcolor = (c) -> {

(Input argument Type) 1) returns value.

return

"color: " + c.getPrice() + "color = " + c.getColor();

Function<Car, String> priceAndColor

= c -> "price :" + c.getPrice() +

"color :" + c.getColor()

Static

String car = priceAndColor.apply(cars.get(0));

sop(stringcar

Stream<String> rows = Files.lines(Paths.get("Data/StockData

"SV.txt"));

rows.map(x -> x.split(","))

.filter(x -> x.length > 3)

.filter(x -> Integer.parseInt(x[1].trim()) > 15)

.forEach(x -> System.out.println(x[0].trim() + " "

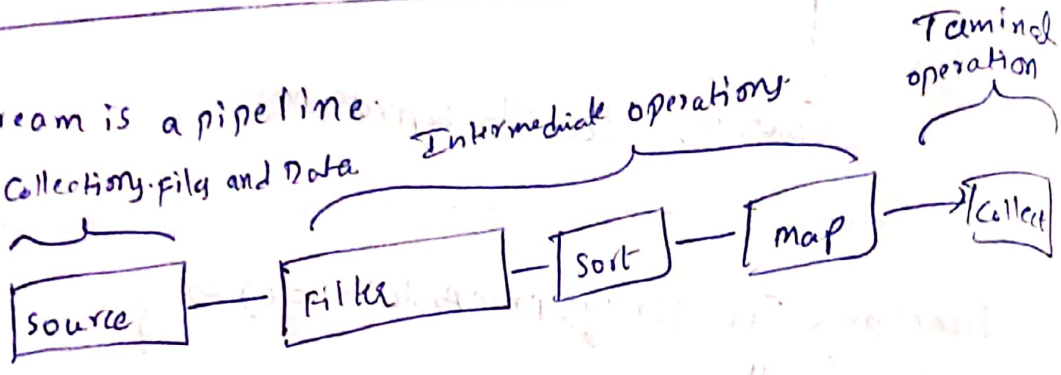
rows.close(); → x[2].trim() + " " x[3].

trim());

Stream API's

Streams make heavy use of lambda expressions.

Stream is a pipeline.



Ex of Intermediate operations:

- filter()
- distinct()
- map()
- skip()
- sorted()

order matters: Filter first, then sort or map etc.

for very large datasets we can use ParallelStream to enable multiple threads to perform operations.

- Zero or more Intermedicle operations are allowed.

Terminal operations:

- forEach() applies the same function to each element.
- collect() saves the elements in to a Collection.
- other options reduce the stream to a single summary element.

.count() . max() ; Sum() ; min() ; Summarystatistics()

class App {

psvm(String [] args) {

Instream } source
.range(1,10) → skips up to 5. operations.
.skip(5) // skip 5 elements of the stream, → Intermediate
.forEach((x) → System.out.print(x)); → terminal operator

System.out.println();

}

1, 2, 3, 4

// Integer Stream with sum

int val = Instream } source.
.range(1,5)

.sum(); → Terminal operations.

Sop(val);

List of data.

Stream.of("Hello", "bottle", "Africa") } source

. sorted() → intermediate operator.

. findFirst() → Terminal

. ifPresent(x) → System.out.println(x));

Stream.of() → function

```
String item[] = { "car", "computer", "toothpaste", "box",
                  "pencil", "tent", "door", "toy" };

Stream.of(items)
    .filter( (x) → x.startsWith("t"))
    .sorted()
    .forEach( x → System.out.print(x + " "));

System.out.println();

// Average of squares of an int array

Array.stream(new int[] {2,4,6,8,10});
    .map(x) → x*x)
    .average()
    .ifPresent(n → System.out.print(n));

System.out.println();
```

```
//Stream from a list, filter

List<String> listOfItems = Arrays.asList("computa", "Box",
                                         "pencil", "car", "Text");

listOfItems.stream()                          ---> making collection of items
    .map(x -> x.toLowerCase())
    .filter(x -> x.startsWith("c"))
    .sorted()
    .forEach(x -> System.out.print(x + ","));


Stream<String> lines = Files.lines(Paths.get("data/
                                   wordFile.txt"));

    lines.sorted()
    .filter(l -> l.length()>6)
    .forEach(x -> System.out.print(x + ", "));

    lines.close();
}
```

```
List<String> words = Files.lines(Paths.get("Data/wordFile.txt"))
    .filter(x -> x.contain("th"))
    .collect(Collectors.toList()); -> converting stream of words to collection.

words.forEach(x -> System.out.print(x + ","));
System.out.println();
```

Intermediate returns stream of data

```
Stream<String> rows = Files.lines(Paths.get("Data/stockDataCSV.txt"));
int rowCount = (int)rows
    .map(x -> x.split(","))
    .filter(x -> x.length > 3)
    .count()
System.out.println(rowCount + "good rows");
rows.close();
```

```
});
```

```
$("#square").animate({
    height: '+=20px'
}, 1000);
```

2nd continue → it continue or

```
$("#square").animate({
    width: '+=20px',   → Increment
    height: '+=20px'
}, 500);
```

Same time ← Increment 20px every time by clicking